

案例源代码
多媒体视频
C编程基础

精通 Linux C 编程

刘学勇　陈建伟　编著

- Linux基础知识
- Linux C语言编程环境
- Linux下的文件编程
- 标准I/O库
- 进程操作
- 进程间通信
- 线程操作
- 网络编程
- 数据库编程
- Linux下的GUI编程
- Linux C编程综合实例

清华大学出版社

精通 Linux C 编程

刘学勇　陈建伟　编著

清华大学出版社
北　京

内 容 简 介

本书系统地介绍了在Linux操作系统下用C语言进行程序设计的方法，并通过列举大量的程序实例，使读者很快地掌握在Linux操作系统下进行C程序开发的方法和技巧，培养开发大型应用程序的能力。

本书内容主要包括Linux基础知识介绍，Linux下的C语言编译器、调试器和程序维护工具的使用方法，Linux下通过C语言进行文件操作和目录操作的方法，标准I/O库函数，进程概念、进程操作以及进程间通信的方法，线程操作，用C语言进行网络编程、数据库编程以及GUI编程的方法等。最后通过一个飞机票网络售票系统的模拟程序演示了Linux C项目开发的方法和流程。

本书结构合理、概念清晰、深入浅出、易于理解，具有很强的实用性，适用于想要系统地学习在Linux系统下进行C语言编程的初级和中级读者阅读，也可作为高等院校计算机相关专业的教材。

图书在版编目(CIP)数据

精通Linux C编程/刘学勇，陈建伟 编著. —北京：清华大学出版社，2009.7

ISBN 978-7-302-20526-5

Ⅰ. 精…　Ⅱ. ①刘…②陈…　Ⅲ. ①Linux操作系统—程序设计②C语言—程序设计　Ⅳ. TP316.89　TP312

中国版本图书馆CIP数据核字(2009)第111081号

责任编辑： 刘金喜　鲍　芳
封面设计： 久久度文化
版式设计： 康　博
责任校对： 胡雁翎
责任印制：
出版发行： 清华大学出版社　　**地　址：** 北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　编：** 100084
社　总　机： 010-62770175　　**邮　购：** 010-62786544
投稿与读者服务： 010-62776969, c-service@tup.tsinghua.edu.cn
质　量　反　馈： 010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　销： 全国新华书店
开　本： 185×260　**印　张：** 29.75　**字　数：** 687千字
附光盘1张
版　次： 2009年7月第1版　**印　次：** 2009年7月第1次印刷
印　数： 1~4000
定　价： 52.00元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：010-62770177 转 3103　　产品编号：



Linux 是当前最流行的操作系统之一。它是由芬兰大学生 Linus 开发的类 Unix 操作系统，它具有系统内核小、稳定性高、可扩展性好、对硬件要求低、网络功能强等特点，现在已经成为成熟的操作系统，并以其良好的稳定性和优异的性能给用户带来了全新的感受，赢得了人们的普遍青睐。

C 语言原是 AT&T 属下的 Bell Labs 的 Dennis Ritchie 为开发 UNIX 操作系统而独立设计并实现的。随着 UNIX 操作系统的广泛流行及微型计算机的普及推广，C 语言作为 Unix 操作系统的孪生兄弟，也广泛地应用于软件开发领域。它的简洁、高效、可移植性等众多优点受到软件开发人员的喜爱，成为最受欢迎的编程语言。

Linux 操作系统同 C 这种具有多平台、移植性好的编程语言的完美结合，为用户提供了一个功能强大的编程环境。掌握 Linux 下的 C 语言编程是学习 Linux 下编程必不可少的一环，本书正是以此为出发点，介绍 Linux 系统下进行 C 语言编程的有关知识。

本书主要针对那些对 Linux 和 C 语言有一定了解，想学习如何在 Linux 系统中使用 C 语言编程的读者。

全书共分 11 章，内容如下：

第 1 章是 Linux 基础知识，介绍了 Linux 的发展、安装以及 Linux 系统的一些常用命令等。

第 2 章介绍了 Linux 下的 C 语言编程环境，主要讨论了 Linux 下 C 语言编程所使用的编辑器、编译器、调试器以及程序管理工具的使用等。

第 3 章是 Linux 下的文件编程，介绍了 Linux 下 C 语言的文件基本 I/O 操作和一些高级操作等。

第 4 章是标准 I/O 库，介绍了 Linux 下基于流的标准 I/O 操作。

第 5 章是进程操作，介绍了 Linux 进程的基本概念以及 Linux 下进程控制的 C 语言编程。

第 6 章是进程间通信，讲述了 Linux 下进程间通信机制以及用 C 语言实现 Linux 下进

程间通信的方法等。

第 7 章是线程操作，介绍了 Linux 线程的基本概念以及线程管理的 C 语言编程等。

第 8 章是网络编程，介绍了 Linux 下网络套接字编程的基本方法。

第 9 章是数据库编程，介绍了数据库的基本概念以及用 C 语言访问 Linux 下 MySQL 数据库的方法。

第 10 章是 Linux 下的 GUI 编程，介绍了 Linux 下 X Window 系统的基本概念、Xlib 编程及 GTK+/GNOME 编程等。

第 11 章通过实现一个飞机票网络售票系统的模拟程序，帮助读者对 Linux 下的 C 程序项目开发的方法和流程有更深一步的认识和提高。

本书语言简练、阐述清晰、实例生动，能很好地帮助读者掌握 Linux 平台下使用 C 语言编程的基本方法和技巧。本书每章都提供了一些完整的应用实例或程序段，这些应用实例可直接在机器上编译。每个应用实例程序都有较强的针对性，说明在程序设计中的方法与技巧。

本书一些重要章节后还附有习题，方便读者学习。

本书由刘学勇、陈建伟编写，戴俊杰、赵强、王玮、张伟娜、夏基平、赵梅、朱运成、王景利、张承伟、罗美云、肖启贵、马丽、陈道允、顾志宏、范士东、施铁良、王琳、吴萍、夏小同和程琳等也参与了本书的资料整理和编写，为本书的问世付出了大量的心血，在此，作者对他们给予的支持和帮助表示最诚挚的感谢。

本书适合那些以前没有接触过 Linux，但又想学习 Linux 下 C 语言编程的读者，有 Linux 操作经验和 C 语言基础，学习起来则更容易一些。

由于作者水平有限，加之时间仓促，书中难免有不妥之处，恳切希望读者予以批评、指正。

作　　者

2009 年 5 月


目录
CONTENTS

CHAPTER 1

Linux 基础知识

Linux 从 1991 年问世到现在，短短十几年的时间已经发展成为功能强大、设计完善的操作系统之一。作为最能体现互联网自由和开放精神的代表，Linux 自诞生以来就以软件源代码开放、可自主开发和高效灵活等特点而迅速得到众多软件开发者的推崇。并且，随着互联网的迅猛发展，Linux 正取代 Windows 成为全球增长最快的操作系统。随着 Linux 应用的普及，Linux 下的软件开发无疑会成为IT 业发展的又一次高潮。本章主要介绍 Linux 的一些基础知识。

1.1 Linux 简 介

本节将简单介绍一下 Linux 的基础知识，包括 Linux 的起源、特点、版本以及它的发展前景等，使读者对 Linux 系统有一个宏观上的了解。

1.1.1 Linux 的起源

Linux 操作系统是 Unix 操作系统的一个克隆版本。Linux 最早是由芬兰人 Linus Torvalds 设计的。

在 Linux 诞生之前，为了教学和研究的需要，阿姆斯特丹 Vrije 大学的计算机科学家 Andrew S. Tanwnbaum 以 Unix 为蓝本开发了 Minix 作为一个教育工具。1991 年初，Linus 开始在一台 386sx 兼容微机上学习 Minix 操作系统。通过学习，他逐渐不能满足 Minix 系统的现有性能，并开始酝酿开发一个新的免费操作系统，很快就在 Minix 新闻组得到了响应。

到了 1991 年的 10 月 5 日，Linus 在 comp.os.minix 新闻组上发布消息，正式向外宣布 Linux 内核系统的诞生(Free minix-like kernel sources for 386-AT)——0.02 版。1991 年 11 月，Linux 0.10 版本推出；0.11 版本随后在 1991 年 12 月推出。当 Linux 非常接近于一种稳定可靠的系统时，Linus 决定将 0.13 版本改称为 0.95 版本。后来在 1994 年 3 月，终于出现了带有独立宣言意味的 Linux 1.0 版本。Linux 1.0 已经是一个功能完善的操作系统了，其内核写得紧凑高效，可以充分发挥硬件的性能，在 4MB 内存的 80386 机器上也表现得非常好。

事实上，Linux 系统是世界各地成千上万志愿者设计和实现的。其目的是建立不受任何商品化软件版权制约的、全世界都能自由使用的类 Unix 操作系统。在 Linux 操作系统的设计过程中，借鉴了很多 Unix 的思想，但源代码是全部重写的。目前 Linux 操作系统可以运行在 x86，Alpha，MIPS，Power Mac，Mach 等类型的计算机上。从功能来看，它既可以作为普通的桌面操作系统，也可以作为中小型的网络操作系统，甚至作为大型网络的操作系统。

1.1.2 Linux 的特点

为什么 Linux 如此备受青睐？就让我们来看一下 Linux 的特点吧。

- 自由软件

Linux 可以说是作为开放源码的自由软件的代表，正是由于这一点，来自全世界的无数程序员参与了 Linux 的修改、编写工作，程序员可以根据自己的兴趣和灵感对其进行修改。这让 Linux 吸收了无数程序员的精华，不断壮大。

- 完全兼容 POSIX 1.0 标准

POSIX 是基于 Unix 的第一个操作系统国际标准，这使得可以在 Linux 下通过相应的模拟器运行常见的 DOS、Windows 的程序。

- 多用户、多任务

Linux 支持多用户，各个用户对于自己的文件设备有自己特殊的权利，保证了各用户之间互不影响。多任务则是现在计算机最主要的一个特点，Linux 可以使多个程序同时独立地运行。

- 良好的用户界面

Linux 向用户提供了两种界面：文本界面和图形用户界面。Linux 的传统用户界面是基于文本的命令行界面，即 shell，它既可以联机使用，又可脱机使用。

Linux 还为用户提供了图形用户界面。它利用鼠标、菜单、窗口、滚动条等元素，给用户呈现一个直观、易操作、交互性强的友好的图形化界面。Linux 的图形用户界面最近几年有很大的改进。在图形用户界面下，几乎可以做全部的工作。

- 支持多种文件系统

Linux 能支持多种文件系统。目前支持的文件系统有：EXT2、EXT、XIAFS、ISOFS、

HPFS、MSDOS、UMSDOS、PROC、NFS、SYSV、MINIX、SMB、UFS、NCP、VFAT、AFFS 等。

- 丰富的网络功能

完善的内置网络是 Linux 的一大特点。 Linux 在通信和网络功能方面优于其他操作系统。其他操作系统不包含如此紧密地和内核结合在一起的连接网络的能力，也没有内置这些联网特性的灵活性。而 Linux 为用户提供了完善的、强大的网络功能。

- 可靠的系统安全

Linux 采取了许多安全技术措施，包括对读写进行权限控制、带保护的子系统、审计跟踪、核心授权等，这为网络多用户环境中的用户提供了必要的安全保障。

- 良好的可移植性

Linux 是一种可移植的操作系统，能够在从微型计算机到大型计算机的任何环境中和任何平台上运行。可移植性为运行 Linux 的不同计算机平台与其他任何机器进行准确而有效的通信提供了手段，不需要另外增加特殊的和昂贵的通信接口。

正是由于以上特点， Linux 在短时间内获得了飞速的发展，成为计算机发展史上的一个奇迹。

1.1.3 Linux 的版本

任何一个软件都有版本号，例如微软的 Windows Vista，Office 2007 等，Linux 也不例外。Linux 的版本号分为两部分：内核(kernel)与发行套件(distribution)版本。

Linux 的内核是系统的心脏，内核包括了 700 多万行代码，是运行程序和管理硬件设备的核心程序。没有内核，就不能运行程序，但内核不是操作系统的全部。Linux 初学者常会把内核版本与发行套件版本弄混了， 实际上内核版本指的是在 Linus 领导下的开发小组开发出的系统内核的版本号。Linux 的每个内核版本使用形式为 x.y.zz-www 的一组数字来表示。其中，x.y 为 Linux 的主版本号，zz 为次版本号，www 代表发行号(注意，它与发行版本号无关)。当内核功能有一个飞跃时，主版本号升级，如 Kernel2.2、2.4、2.6 等。内核增加了少量补丁时，常常会升级次版本号，如 Kernle2.6.15、2.6.20 等。当然还有更复杂的版本号系统，如 2.6.20-32 等。通常 y 若为奇数，表示此版本为测试版，系统会有较多 bug，主要用途是提供给用户测试。随着每一次系统小 bug 的修正，zz 会增加。编写本书时，Linux 的内核版本号是 2.6.22-www(主版本号 2.6 表明它是可以使用的稳定版本)。

一般而言，一个基本的 Linux 只是包含了 Linux 核心(kernel)和 GNU 软件的一些基层系统软件和实用工具(utilities)，这样一个操作系统仅仅能够让那些 Linux 专家完成一些很基本的系统管理任务，若要满足普通用户的办公或基于视窗的应用开发等需要，则还需要在系统中加入 XFree86 视窗系统、GNOME 或 KDE 桌面环境以及相应的办公应用软件(如 OpenOffice)等。因此一些组织或厂家将 Linux 系统内核与 GNU 软件(系统软件和工具)整合起来，并提供一些安装界面和系统设定与管理工具，这样就构成了一个发行套件，例如最

常见的 Slackware，Red Hat，Debian，Ubuntu 等。实际上发行套件就是 Linux 的一个大软件包而已，通常包括 C 语言及 C++的编译器、Perl 脚本解释程序、Shell 命令解释器、图形用户界面 X 窗口系统、X Server 以及众多的应用程序。相对于内核版本，发行套件的版本号随发布者的不同而不同，与系统内核的版本号是相对独立的。因此把 Red Hat，Slackware 等直接说成是 Linux 是不确切的，它们是 Linux 的发行版本，更确切地说，应该叫做“以 Linux 为核心的操作系统软件包”。根据 GPL 准则，这些发行版本虽然都源自一个内核，并都有各自的贡献，但都没有自己的版权。Linux 的各个发行版本，都是使用 Linus 主导开发并发布的同一个 Linux 内核，因此在内核层不存在兼容性问题。至于每个版本都不一样的感觉，只是在发行版本的最外层才有所体现，而绝不是本身，也不是内核不统一或不兼容。

目前 Linux 的发行版很多，其中比较流行的国外版本有：Red Hat、Ubuntu、Slackware、Debian、SuSE 和 Mandrake 等；国内的有 Xteam Linux、Turbo Linux 和红旗 Linux 等。

1.1.4 Linux 的发展前景

Linux 以其独立、开放、安全、免费、强大的网络功能等特点，已在各个行业得到了广泛的应用，同时，Linux 的嵌入式和中间件方式具有优秀的移植性，利用 Linux 系统来进行软件开发已经成为一种趋势。可以想象，Linux 的发展前景非常可观。

1.2 Linux 的 安 装

“工欲善其事，必先利其器”。要进行 Linux 下的编程，必须先安装 Linux 系统。在安装 Linux 之前，至少需要考虑两个问题：选择哪个 Linux 发行版本，硬件是否能够支持 Linux。

1.2.1 发行版本的选择

Linux 有多个发行版本可供选择。Ubuntu 是一个相对较新的发行版，它的出现可能改变了许多潜在用户对 Linux 的看法，笔者的笔记本电脑使用的便是 Ubuntu。也许，从前人们会认为 Linux 难以安装、难以使用，但是，Ubuntu 出现后，这些都成为了历史。Ubuntu 以 Debian Linux为基础，针对桌面应用进行了优化。简单而言，Ubuntu 就是一个拥有 Debian 所有的优点，以及自己所加强的优点的近乎完美的 Linux 操作系统。Ubuntu 的安装非常人性化，只需按照提示一步步进行，和安装 Windows 操作系统同样简便！并且，Ubuntu 被誉为对硬件支持最好最全面的 Linux 发行版之一，许多在其他发行版上无法使用，或者

默认配置时无法使用的硬件，在 Ubuntu 上都能轻松搞定。并且，Ubuntu 采用自行加强的内核(kernel)，安全性方面更上一层楼。并且，Ubuntu 默认不能直接 root 登录，而必须从第一个创建的用户通过 su 或 sudo 来获取 root 权限(这也许不太方便，但无疑增加了安全性，避免用户由于粗心而损坏系统)。Ubuntu 的版本周期为六个月，弥补了 Debian 更新缓慢的不足。因此在本书中以 Ubuntu 作为 Linux 程序的开发平台。写作本书时，Ubuntu 的最新发行版为 7.10，读者可以从 Ubuntu 官方站点 http://www.ubuntulinux.org/免费下载发行版的 ISO 文件，然后通过刻录机将 ISO 文件刻录到光盘上。另外也可以向官方申请免费的 Ubuntu 安装光盘，只要在官方网页注册一下，填写相关的地址信息之后，就可以申请，光盘会从荷兰寄过来，大约需要 4~6 周时间。免费申请光盘的官方网址为 https://shipit.ubuntu.com/。

1.2.2 基本的硬件要求

硬件需求应该是安装前最重要的考虑事项之一，虽然 Ubuntu 安装需要的硬件条件并不高，但是最好能提供满足条件的硬件，这样才能达到计划中的性能表现。

欲了解 Ubuntu 安装时的最小硬件需求，可以参考安装 CD/DVD 中的 RELEASE NOTES 文件，或下面的网页：

http://www.ubuntu.com/download/releasenotes

http://help.ubuntu.com

就当前的最新版本而言，以下列出的是安装 Ubuntu 系统最低硬件要求：

- Intel 386 以上处理器的 CPU。
- 至少 32MB 内存，推荐使用 64MB 以上内存。
- 至少 1GB 以上的硬盘空间。
- VGA 显卡。
- CD-ROM 光驱。
- 声卡、网卡等其他设备。

本书写作时，计算机主流配置一般是双核 CPU、1GB 内存，160GB 以上硬盘，因此远远满足 Linux 对系统硬件的最低要求。

1.2.3 安装步骤

(1) 准备好了 Linux 的安装光盘之后，将光盘放入光驱，进入 BIOS 设置，将启动顺序中的第一项设置为“从光盘启动”，保存后退出。若一切无误，利用光盘启动后，系统会弹出如图 1-1 所示的 Ubuntu Live CD 的启动画面。



图 1-1　Ubuntu Live CD 的启动画面

(2) 进入 Ubuntu Live CD 启动画面后，按 F2 键选择语言，可以根据自己的喜好选择，在此选择简体中文，如图 1-2 所示。



图 1-2　中文语言安装界面

(3) 在安装界面中有多个选项供用户选择，这里选择第 1 个“启动或安装 Ubuntu”，可以直接按 Enter 键。随后系统会运行位于 CD 上的一个最小 Ubuntu 桌面系统，如图 1-3 所示。



图 1-3　最小 Ubuntu 桌面系统

(4) 在桌面上有一个安装图标，双击运行它。这时会出现安装欢迎界面，如图 1-4 所示。

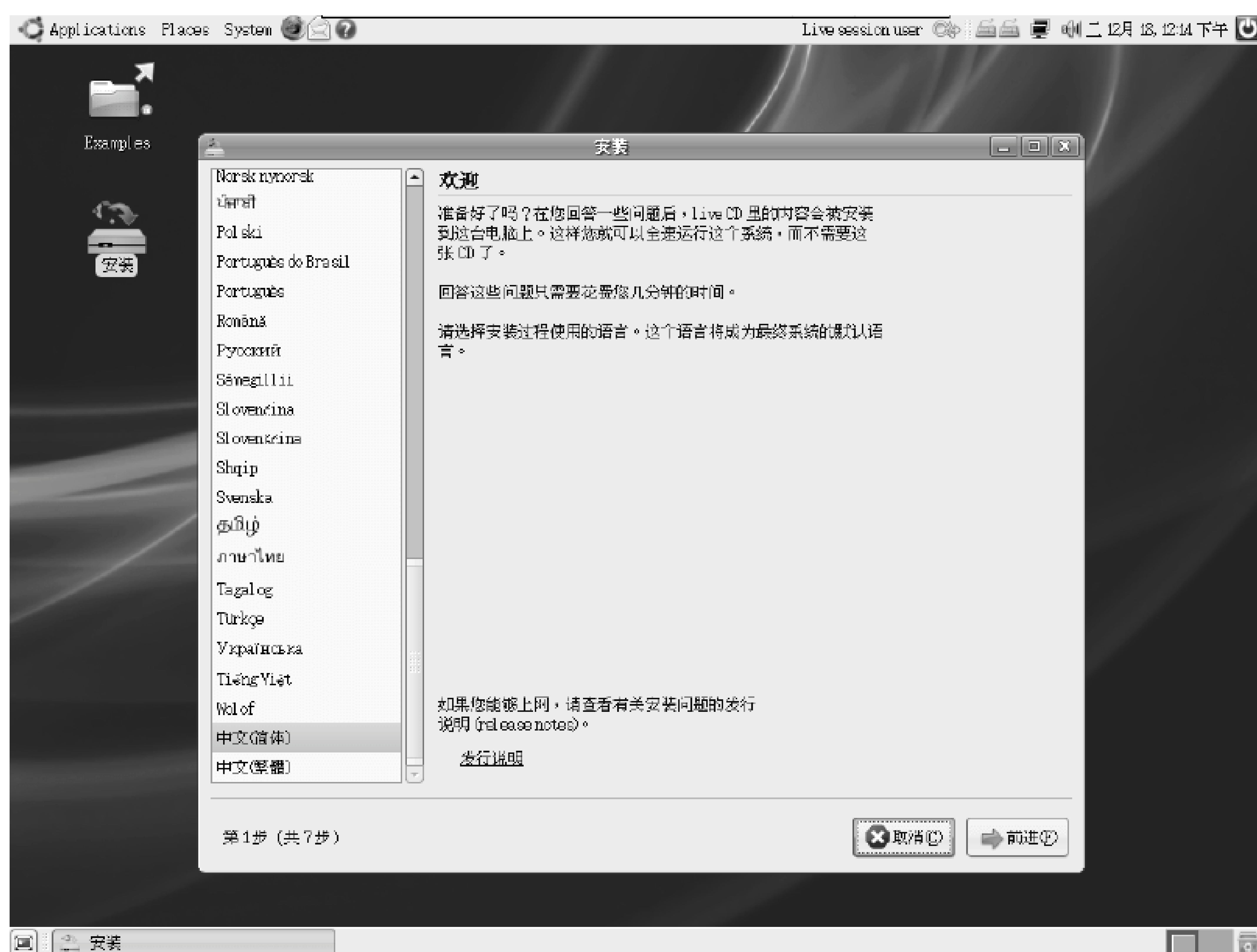


图 1-4　安装欢迎界面

(5) 单击“前进”按钮，进入时区选择界面，如图 1-5 所示，一般选择 Shanghai 就可以了。



图 1-5　时区选择界面

(6) 单击“前进”按钮，进入键盘布局选择界面，如图 1-6 所示，选择 U.S.English。

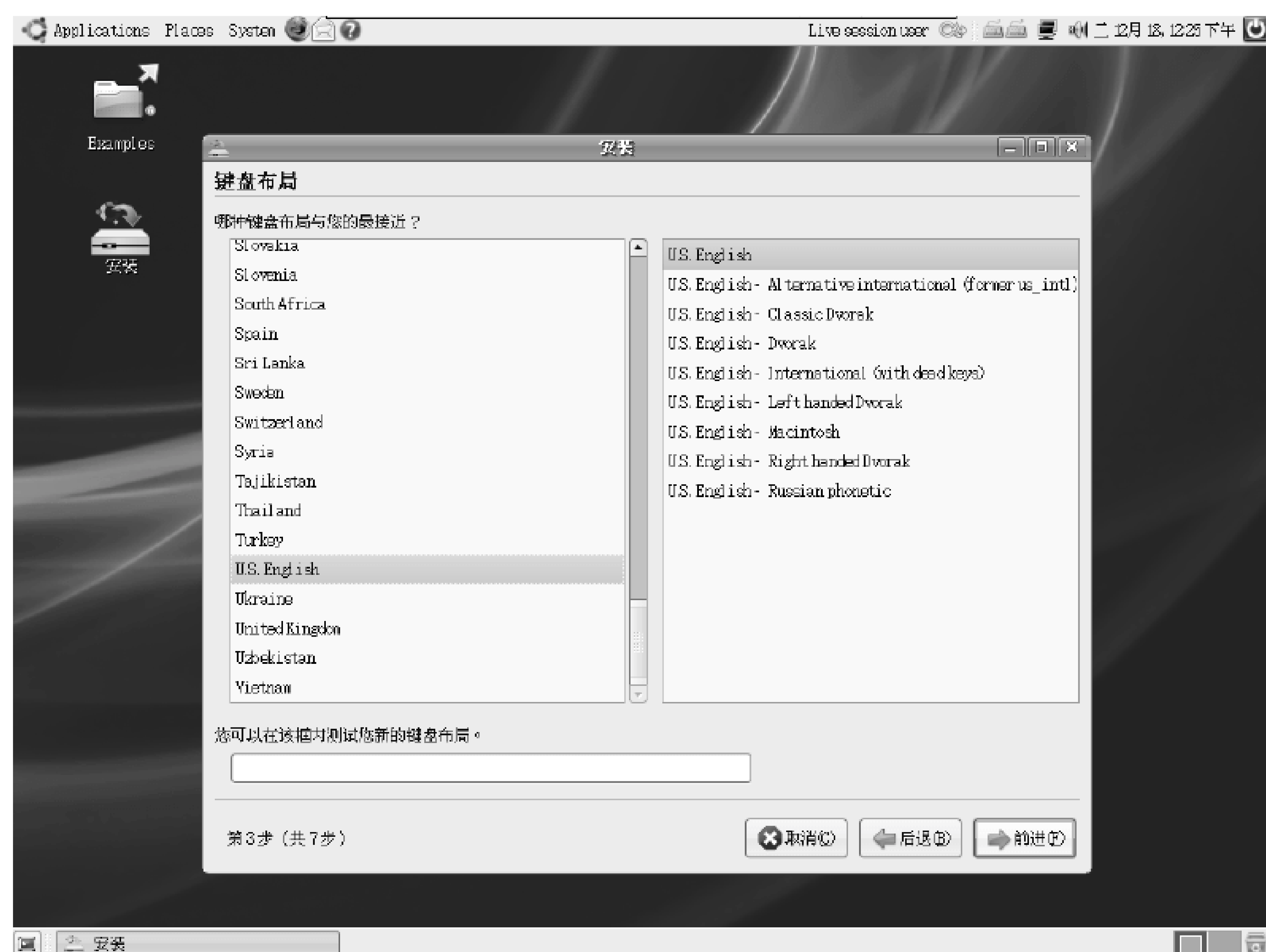


图 1-6　键盘布局选择界面

(7) 单击“前进”按钮，进入“准备硬盘空间”界面，如图 1-7 所示。在这个界面中，共有两种选择，向导和手动创建分区。向导是由系统默认划分硬盘分区，并自动选择根文件系统挂载点与交换分区。此选项会删除硬盘上的所有数据，一般适用于全新的系统。如果硬盘上已经有了较多的资料，则应选择手动分区。在此选择“手动”。



图 1-7　准备硬盘空间界面

(8) 单击“前进”按钮，进入“准备分区”界面，如图 1-8 所示。



图 1-8　准备分区界面

(9) 单击 New partition table 按钮，创建分区。此时会出现一个警告，提示用户创建新的分区表，现有的所有分区都将被删除。选择“返回”不进行分区操作，选择“继续”则进行分区操作，在此选择“继续”按钮。

(10) 单击 New partition 按钮，弹出“创建新分区”对话框，如图 1-9 所示。



图 1-9　创建新分区对话框

(11) 在空白分区中新建分区，具体分区大小可以按照自己的喜好，至少要有 2 个分区，一个 swap 交换分区，一个根分区。

(12) 创建好新分区之后，单击“前进”按钮，进入创建登录用户界面，如图 1-10 所示。



图 1-10　创建登录用户界面

(13) 输入新用户的相关信息后，单击“前进”按钮，进入“准备安装”界面，如图 1-11 所示。在这个界面中主要是确认此前输入的相关信息是否正确。如果发现有误，可以选择“后退”继续修改。



图 1-11 准备安装界面

(14) 单击 Install 按钮，系统开始从光盘往硬盘复制文件。复制文件结束后，取出光盘，重启系统，系统安装完成。以后可以根据需要，对系统进行一些必要的设置。关于这方面的详细内容，读者可参考其他相关资料。

1.3 Linux 系统的常用命令

在 Linux X Window 图形界面下，可以完成大部分工作，然而，许多 Linux 功能在命令行界面下要比在 X Window 图形界面下完成得更快。本节先介绍 Shell 的概念，然后介绍 Shell 的命令行操作界面，最后介绍常用的 shell 命令。

1.3.1 了解 Shell

在学习 Linux 常用命令之前，首先应该了解一下这些命令的运行环境。这个环境就是 Shell，它是一种命令解释器，在用户和操作系统之间提供了一个交互接口。用户在命令行

输入命令，然后 Shell 对该命令进行解释并将它作为指令代码发送给操作系统。Shell 的这种解释功能提供了许多高级特性。例如，Shell 有一套能产生文件名的通配符，Shell 不仅能够对输入、输出重定向，而且能够在后台执行命令，从而使用户同时可以进行其他任务的操作。在 Linux 操作系统中有许多可选的 Shell。每种 Shell 提供不同的特性和功能，大多数 Shell 有自己的脚本语言，使用脚本语言可以建立复杂的自动执行程序。由于 Ubuntu 的默认 Shell 是 Bash，因此本节介绍的命令也是在 Bash Shell 环境下运行的。

1.3.2 进入 Shell 命令行界面

Shell 是终端下的用户操作界面。Linux 终端也称为虚拟控制台，如果读者使用过 Unix，那么对此一定不会陌生。显示器和键盘合称为终端，因为它们可以对系统进行控制，所以又称为控制台，一台计算机的输入/输出设备就是一个物理控制台。如果在一台计算机上用软件的方法实现多个互不干扰、独立工作的控制台界面，就是实现了多个虚拟控制台。Linux 终端采用字符命令行方式工作，用户通过键盘输入命令，通过 Linux 终端对系统进行控制。通常情况下，Linux 默认启动 6 个虚拟终端，如果选择直接启动 X Window 启动方式，则 X Window 是在第 7 个虚拟终端上。因此 Ubuntu 的 X Window 就是在第 7 个虚拟终端上。

在 X Window 图形界面中，按 Alt+Ctrl+Fn(n=1~6)键或者 Ctrl+Alt+F7 键就可以进入控制台字符操作界面。在控制台字符操作界面下，按 Alt+Fn(n=1~6)键可以切换到其他字符操作界面，在控制台字符操作界面下，可以按 Alt+F7 键，返回 X Window 图形界面。此外，在 X Window 图形界面中，Linux 还提供了图形终端。在 Ubuntu 中进入图形终端的方法是：选择“应用程序”|“附件”|“终端”命令，打开图形终端窗口，如图 1-12 所示。

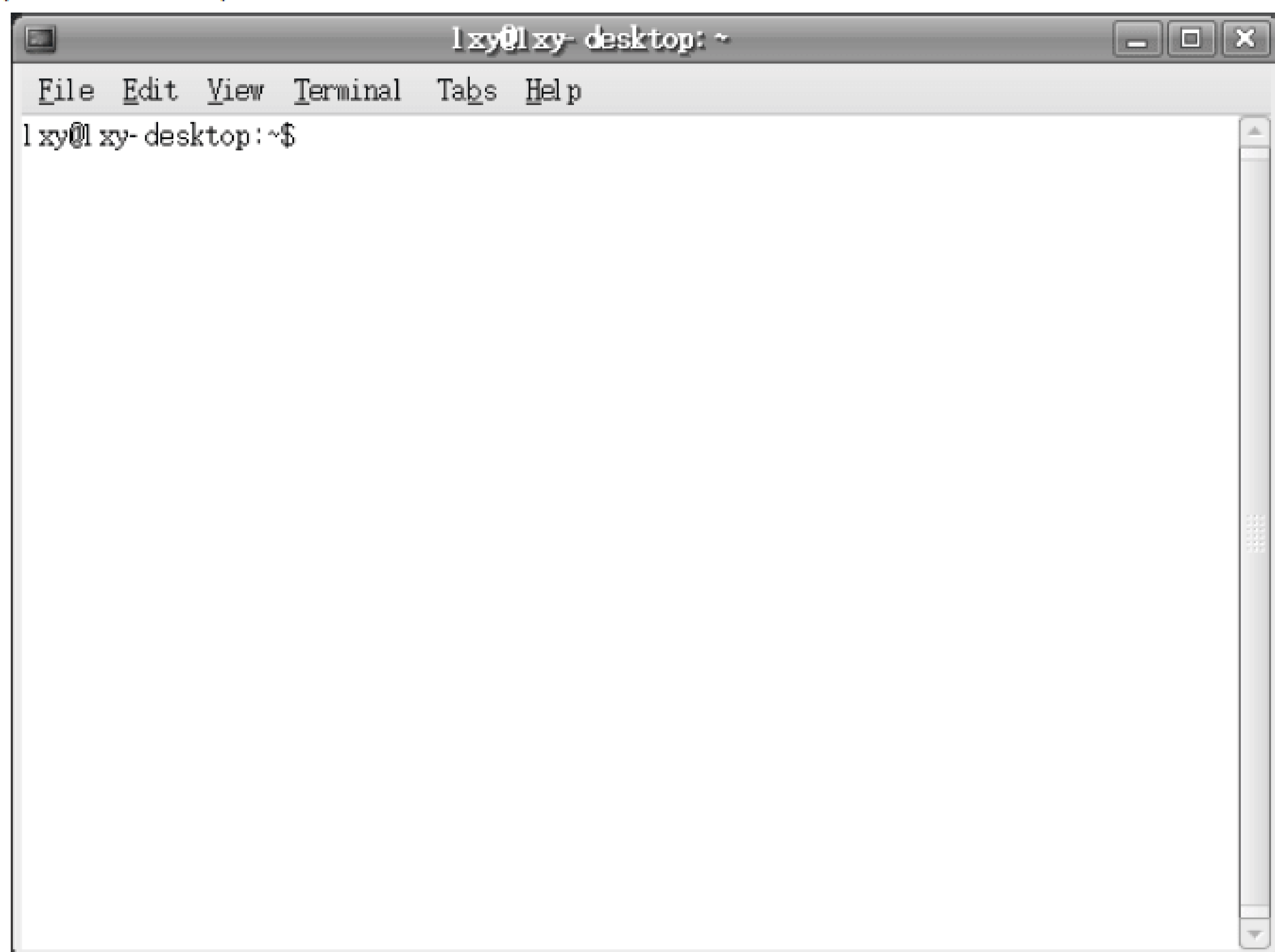


图 1-12　Ubuntu 图形终端窗口

其中 lxy@lxy-desktop:~$被称为 Shell 的命令提示符；“@”前面的字符表示登录的用户名，“@”后面的字符表示登录的计算机名。所以提示符 lxy@lxy-desktop:~$表示的意思

为登录的用户是 lxy；登录的计算机是 lxy-desktop。另外，$是普通用户的提示符，而#是超级用户提示符。

出现命令提示符后，就可在光标处输入命令。每条命令输入完毕后，必须按 Enter 键才会执行。如果输入的命令中有某个字符需要删除或修改，可以用左右方向键将光标移到要修改字符的后面或前面，再按 BackSpace 或 Delete 键删除，然后再输入正确的字符。如果想调用以前输入过的命令，可用向上或向下的方向键进行选择。如在命令提示符后输入 date，按 Enter 键，系统就会显示当前的日期和时间，如下所示：

```
lxy@lxy-desktop:~$ date
2007 年 12 月 20 日 星期四 16:24:28 CST
lxy@lxy-desktop:~$
```

第一行是我们输入的 date 命令，第 2 行是系统对命令的响应，这里显示的是当前日期。命令响应完成后，系统又返回到等待输入命令的状态，如第 3 行所示。

又如询问当前有哪些用户挂在系统里。命令及响应如下所示：

```
root@lxy-desktop:~# who
lxy        tty7        2007-12-21 14:48 (:0)
lxy        pts/0       2007-12-21 15:15 (:0.0)
```

显示结果可以看出，当前系统中只有一个用户 lxy，分别在 tty7 虚拟终端和 pts/0 虚拟终端登录，其中 tty7 即为 X Window 图形界面所在的虚拟终端，而 pts/0 则为 X Window 下的图形终端。

在字符终端下和在图形终端下的操作是没有区别的，读者可以根据自己的兴趣选择相应的终端操作类型。

1.3.3 文件操作命令

在 Linux 中，文件是用来存储信息的基本结构，它是存储在某种介质上的一组信息的集合。文件名是文件的标识，它包含字母、数字、下划线和句点组成的字符串。Linux 系统中有三种基本的文件类型：普通文件、目录文件和设备文件。

普通文件是用户最常使用的文件，它可分为文本文件和二进制文件。

目录用于管理和组织系统中的大量文件。在 Linux 系统中，目录以文件的形式存在，目录文件存储了一组相关文件的位置、大小等与文件有关的信息。目录文件简称为目录。

Linux 系统把每一个 I/O 设备都看成一个文件，用处理普通文件的方法处理它们，这样可以使文件与设备的操作尽可能统一。从用户的角度看，I/O 设备的使用和一般文件的使用一样，不必了解 I/O 设备的细节。

文件操作是 Linux 系统里最基本也是最常用的操作，本节列举了 Linux 下经常要执行的一些普通文件操作命令。

1.3.3.1 文件显示命令

显示指定工作目录中所包含的内容的指令是 ls，要说明的是 ls 命令列出文件的名字，而不是文件的内容。该命令的使用方式如下：

```
ls [选项]  [文件目录列表]
```

ls 命令中的常用选项如表 1-1 所示。

表 1-1 ls 命令选项说明

选　项	说　明
-a	列出目录下的所有文件，包括以“.” 开头的隐含文件
-b	把文件名中不可输出的字符用反斜杠加字符编号(就像在 C 语言里一样)的形式列出
-c	输出文件的 i 节点的修改时间，并以此排序
-d	将目录像文件一样显示，而不是显示其下的文件
-e	输出时间的全部信息，而不是输出简略信息
-f-U	对输出的文件不排序
-i	输出文件的 i 节点的索引信息
-k	以 k 字节的形式表示文件的大小
-l	列出文件的详细信息
-m	横向输出文件名，并以“，”作分隔符
-n	用数字的 UID,GID 代替名称
-o	显示文件的除组信息外的详细信息
-p -F	在每个文件名后附上一个字符以说明该文件的类型，“*”表示可执行的普通文件；“/”表示目录；“@”表示符号链接；“|”表示 FIFOs；“=”表示套接字(sockets)
-q	用“?”代替不可输出的字符
-r	对目录反向排序
-s	在每个文件名后输出该文件的大小
-t	以时间排序
-u	以文件上次被访问的时间排序
-x	按列输出，横向排序
-A	显示除 “.”和“..”外的所有文件
-B	不输出以 “~”结尾的备份文件
-C	按列输出，纵向排序
-G	输出文件的组的信息
-L	列出链接文件名而不是链接到的文件

(续表)

选　项	说　明
-N	不限制文件长度
-Q	把输出的文件名用双引号括起来
-R	列出所有子目录下的文件
-S	以文件大小排序
-X	以文件的扩展名(最后一个 “.” 后的字符)排序
-1	一行只输出一个文件
--color=no	不显示彩色文件名

Linux 支持多种文件类型，每一类用一个字符来表示，其说明如表 1-2 所示。

表 1-2　Linux 文件类型说明

文 件 类 型	说　明
-	常规文件
d	目录
b	块特殊设备
c	字符特殊设备
p	有名管道
s	信号灯
m	共享存储器

文件类型的字符表示文件的权限，权限由三个字符串组成，这三个字符串分别表示：该文件所有者的权限、组中其他人的权限和系统中其他人的权限；每个字符串又由三个字符组成，依次表示对文件的读(用字符 r 表示)、写(用字符 W 表示)和执行权限(用字符 x 表示)。当用户没有相应的权限时，该权限的对应位置用短线“-”来表示。 例如：

```
drwxr-x---
```

表示的含义是：d 表示这条信息是目录；目录拥有者的权限是 rwx(表示有读、写和执行权限)；组中其他人对该目录的权限是 r-x(表示有读和执行权限，没有写权限)，系统中其他人对该目录的权限是---(表示读、写和执行权限都没有)。

例 1-1　查看当前目录的内容，命令及响应如下所示：

```
root@lxy-desktop:~/test# ls
2.6.22-vmhgfs-55017.tar.bz2     file2.txt~   viewarticle.php.html
file1.txt                       file3.txt    vmhgfs.tar
file1.txt~                      file4.txt    vmxnet-2.6.22-rc1-vmws6(2).tar
```

```
file2.txt                    file4.txt~   vmxnet.tar
```

从显示结果可以看出，当前目录下有 12 个文件或目录。

例 1-2 查看根目录下 usr 子目录下的内容，命令及响应如下所示：

```
root@lxy-desktop:~/test# ls /usr
bin   games   include   lib   local   sbin   share   src   X11R6
```

从显示结果可以看出，/usr 目录下共有 9 个文件或目录。

例 1-3 用长格式查看根目录下 usr 子目录下的内容，命令及响应如下所示：

```
root@lxy-desktop:~/test# ls -l /usr
drwxr-xr-x    2 root root      36864   2007-12-21   14:46 bin
drwxr-xr-x    2 root root       4096   2007-12-12   22:41 games
drwxr-xr-x    38 root root      4096   2007-12-13   21:34 include
drwxr-xr-x    150 root root    36864   2007-12-21   14:46 lib
drwxr-xr-x    10 root root      4096   2007-10-16   07:17 local
drwxr-xr-x    2 root root      12288   2007-12-21   14:46 sbin
drwxr-xr-x    280 root root    12288   2007-12-13   21:09 share
drwxrwsr-x    4 root src        4096   2007-10-16   07:22 src
drwxr-xr-x    3 root root       4096   2007-10-16   07:19 X11R6
```

用长格式查看目录内容，每行表示一个文件或目录的信息，每行信息依次表示：文件类型与权限、连接数、文件属主、文件属组、文件大小、建立或最近修改的时间、名字。例如，上述命令响应的第一行第一列表示 bin 是一个目录，拥有者对该目录的权限是读写和执行。组中其他人对该目录的权限是读和执行，系统中其他人对该目录的权限是读和执行。第 2 列表示 bin 的硬链接数是 2。通过 ln 创建链接时，该数值会加 1。关于 ln 命令将在后面进行详细讲解。第 3 列表示该目录的所有者是 root，第四列表示 bin 所属的工作组是 root，第 5 列表示 bin 目录的大小是 36864 bytes，第 6 列表示 bin 目录建立的时间是 2007 年 12 月 21 日 14 时 46 分。

1.3.3.2 查找文件

在 Linux 系统中查找文件的命令通常为 find 命令。find 命令可在使用、管理 Linux 系统中方便地查找所需要的指定文件。

find 命令的使用方式如下：

```
find [目录列表]  [匹配标准]
```

在命令格式中有两个参数，说明如下：

目录列表：希望查询文件或文件集的目录列表，目录间用空格分隔。

匹配标准：希望查询的文件的匹配标准或说明。详细的匹配标准如表 1-3 所示。

表 1-3 find 命令匹配标准说明

选　项	说　明
-amin n	查找系统中最后 N 分钟访问的文件
-atime n	查找系统中最后 n*24 小时访问的文件
-cmin n	查找系统中最后 N 分钟被改变状态的文件
-ctime n	查找系统中最后 n*24 小时被改变状态的文件
-empty	查找系统中空白的文件，或空白的文件目录，或目录中没有子目录的文件夹
-false	查找系统中总是错误的文件
-fstype type	查找系统中存在于指定文件系统的文件，例如：ext2
-gid n	查找系统中文件数字组 ID 为 n 的文件
-group gnome	查找系统中文件属于 gnome 文件组，并且指定组和 ID 的文件
-daystart	测试系统从今天开始 24 小时以内的文件，用法类似-amin
-depth	使用深度级别的查找过程方式，在某层指定目录中优先查找文件内容
-follow	遵循通配符链接方式查找；另外，也可忽略通配符链接方式查询
-maxdepth levels	在某个层次的目录中按照递减方法查找
-mount	不在文件系统目录中查找

例 1-4　从根目录“/”下开始，查找 xorg.conf 文件，命令及响应如下所示：

```
root@lxy-desktop:~/test# find / -name xorg.conf -print
/usr/share/xresprobe/xorg.conf
/etc/X11/xorg.conf
```

从显示结果可以看出，分别在/usr/share/xresprobe 和/etc/X112 个子目录中找到 xorg.conf 文件。

例 1-5　当要查找某个文件时，不知道该文件的全名，只知道这个文件包含几个特定的字母，可以借助于通配符“*”、“?”来查找相应的文件。其中，“*”表示任意字符及其组合，“?”表示任意单一字符。例如，从根目录下，查找以 file 字符串开始，后面的字符串为任意字符的 txt 文件，可使用如下所示的命令查找：

```
root@lxy-desktop:/# find / -name file*.txt -print
/usr/share/doc/gname-keyring/file-format.txt
/home/lxy/file3.txt
/home/lxy/file4.txt
/home/lxy/test/file3.txt
/home/lxy/test/file4.txt
/home/lxy/test/file2.txt
/home/lxy/test/file1.txt
/home/lxy/test/file4.txt
```

从显示结果可以看出，包含 file 字符串的文件都被找到，它们分别位于/usr/share/doc/gnome-keyring/、/home/lxy/和/home/lxy/test/目录下。

1.3.3.3 显示文本文件内容

显示文本文件内容的命令是 cat 命令，用来将文件的内容显示到终端上。该命令的使用方式如下：

```
cat  [选项]  文件列表
```

cat 命令中的选项说明如表 1-4 所示。

表 1-4　cat 命令选项说明

选　项	说　明
-v	用一种特殊形式显示控制字符，LFD 与 TAB 除外。加了-v 选项后，-T 和 -E 选项将起作用。其中：-T 将 TAB 显示为“ù I”。该选项需要与 -v 选项一起使用。即如果没有使用 -v 选项，则这个选项将被忽略。-E 在每行的末尾显示一个 $ 符。该选项需要与 -v 选项一起使用
-u	输出不经过缓冲区
-A	等于 –vET
-t	等于 –vT
-e	等于 –vE
-n	在文件的每行前面显示行号

例 1-6　显示 file1.txt 文件内容，并在每行前面显示行号，其命令及响应如下所示：

```
root@lxy-desktop:/home/lxy/test# cat -n file1.txt
1   Linux C 编程指南
2
3
4
```

从显示结果可以看出 file1.txt 共有 4 行，其中第一行的内容为：Linux C 编程指南。后 3 行的内容为空白。

cat 命令功能之二是用来将 2 个或多个文件连接起来。例如把文件 file1.txt 和文件 file2.txt 内容合并起来，放入文件 file4 中，命令及响应如下所示。

```
root@lxy-desktop:/home/lxy/test# cat file1.txt
Linux C 编程指南
root@lxy-desktop:/home/lxy/test# cat file2.txt
cat 命令使用说明
root@lxy-desktop:/home/lxy/test# cat file1.txt file2.txt>file4.txt
```

```
root@lxy-desktop:/home/lxy/test# cat file4.txt
Linux C 编程指南
cat 命令使用说明
```

上面的第一个命令是显示 file1.txt 的内容，第 2 个命令是显示 file2.txt 的内容，第 3 个命令是把 file1.txt 和 file2.txt 的内容合并起来，放入文件 file4.txt 中，最后一个命令是显示 file4.txt 文件内容。从显示结果可以看出，file1.txt 和 file2.txt 的内容已经合并到 file4.txt 中去了。

1.3.3.4 查找文件内容

查找文件内容的命令是 grep 命令。该命令的使用方式如下：

```
grep [选项] [查找模式] [文件名 1，文件名 2，…]
```

grep 命令中的选项说明如表 1-5 所示。

表 1-5 grep 命令选项说明

选　项	说　明
-E	每个模式作为一个扩展的正则表达式对待
-F	每个模式作为一组固定字符串对待(以新行分隔)，而不作为正则表达式
-b	在输出的每一行前显示包含匹配字符串的行在文件中的字节偏移量
-c	只显示匹配行的数量
-i	比较时不区分大小写
-h	在查找多个文件时，指示 grep 不要将文件名加入到输出之前
-l	显示首次匹配串所在的文件名并用换行符将其隔开。当在某文件中多次出现匹配串时，不重复显示此文件名
-n	在输出前加上匹配串所在行的行号(文件首行行号为 1)
-v	只显示不包含匹配串的行
-x	只显示整行严格匹配的行
-e expression	指定检索使用的模式。用于防止以“-”开头的模式被解释为命令选项
-f expfile	从 expfile 文件中获取要搜索的模式，一个模式占一行

例 1-7 在文件 text 中查找包含 linux 的行，命令及响应如下所示：

```
root@lxy-desktop:/home/lxy/test# grep linux -n text
1:linux C 编程指南
```

从显示结果可以看出，在 text 的文件第 1 行中找到了包含 linux 的字符串。

1.3.3.5 排序命令

sort 命令的功能是对文件中的各行进行排序。sort 命令有许多非常实用的选项，这些选项最初是用来对数据库格式的文件内容进行各种排序操作的。实际上，sort 命令可以被认为是一个非常强大的数据管理工具，用来管理内容类似数据库记录的文件。sort 命令将逐行对文件中的内容进行排序，如果两行的首字符相同，该命令将继续比较这两行的下一字符，如果还相同，将继续进行比较。 该命令的使用方式如下：

```
sort  [选项]  文件
```

sort 命令对指定文件中所有的行进行排序，并将结果显示在标准输出上。如不指定输入文件或使用“-”，则表示排序内容来自标准输入。

sort 排序是根据从输入行抽取的一个或多个关键字进行比较来完成的。排序关键字定义了用来排序的最小的字符序列。默认情况下以整行为关键字按 ASCII 字符顺序进行排序。

sort 命令中的选项说明如表 1-6 所示。

表 1-6 sort 命令选项说明

选　项	说　明
-m	若给定文件已排好序，合并文件
-c	检查给定文件是否已排好序，如果它们没有排好序，则打印一个出错信息，并以状态值 1 退出
-u	对排序后认为相同的行只留其中一行
-o	输出文件，将排序输出写到输出文件中而不是标准输出，如果输出文件是输入文件　之一，sort 先将该文件的内容写入一个临时文件，然后再排序和写输出结果
-d	按字典顺序排序，比较时仅字母、数字、空格和制表符有意义
-f	将小写字母与大写字母同等对待
-l	显示首次匹配串所在的文件名并用换行符将其隔开。当在某文件中多次出现匹配串时，不重复显示此文件名
-I	忽略非打印字符
-M	作为月份比较：“JAN” ＜ “FEB” ＜...＜ “DEC”
-r	按逆序输出排序结果
+posl -pos2	指定一个或几个字段作为排序关键字，字段位置从 pos1 开始，到 pos2 为止(包括 pos1，不包括 pos2)。如不指定 pos2，则关键字为从 pos1 到行尾。字段和字符的位置从 0 开始
-b	在每行中寻找排序关键字时忽略前导的空白(空格和制表符)
-t separator	指定字符 separator 作为字段分隔符

例 1-8 用 sort 命令对 text 文件中各行排序后输出其结果的命令及响应如下所示：

```
lxy@lxy-desktop:~$ cat  text
```

```
vegetable soup
fresh vegetables
fresh fruit
lowfat milk
lxy@lxy-desktop:~$ sort    text

fresh fruit
fresh vegetables
lowfat milk
vegetable soup
```

从显示结果可以看出，在原文件的第二、三行上的第一个单词完全相同，该命令将从它们的第二个单词 vegetables 与 fruit 的首字符处继续进行比较。

例 1-9 用户可以保存排序后的文件内容，或把排序后的文件内容输出至打印机，如把排序后的文件内容保存到名为 result 的文件中的命令及响应如下所示：

```
lxy@lxy-desktop:~$ sort text>result
lxy@lxy-desktop:~$ cat result

fresh fruit
fresh vegetables
lowfat milk
vegetable soup
```

从显示结果可以看出 result 中的内容已经是排序后的 text 文件内容。

1.3.3.6 比较文件内容的命令

1. comm 命令

如果想对两个有序的文件进行比较，可以使用 comm 命令。该命令的使用方式如下：

```
comm [-123 ] file1 file2
```

该命令是对两个已经排好序的文件进行比较。其中 file1 和 file2 是已排序的文件。comm 读取这两个文件，然后生成三列输出：仅在 file1 中出现的行；仅在 file2 中出现的行；在两个文件中都存在的行。如果文件名用“-”，则表示从标准输入读取。选项 1、2 或 3 抑制相应的列显示。

例如：comm -12 只显示在两个文件中都存在的行；
comm -23 只显示在第一个文件中出现而未在第二个文件中出现的行；
comm -123 什么也不显示。

2. diff 命令

diff 命令的功能为逐行比较两个文本文件，列出其不同之处。它对给出的文件进行系

统的检查，并显示出两个文件中所有不同的行，不要求事先对文件进行排序。该命令的使用方式如下：

```
diff [选项] file1 file2
```

该命令告诉用户，为了使两个文件 file1 和 file2 一致，需要修改它们的哪些行。如果用 "- "表示 file1 或 file2，则表示标准输入。如果 file1 或 file2 是目录，那么 diff 将使用该目录中的同名文件进行比较。 通常输出由下述形式的行组成：

```
n1 a n3，n4
n1，n2 d n3
n1，n2 c n3，n4
```

字母(a、d 和 c)之前的行号(n1、n2)是针对 file1 的，其后面的行号(n3、n4)是针对 file2 的。字母 a、d 和 c 分别表示附加、删除和修改操作。

在上述形式的每一行的后面跟随受到影响的若干行，以"＜"打头的行属于第一个文件，以"＞"打头的行属于第二个文件。

diff 能区别块和字符设备文件以及 FIFO(管道文件)，不会把它们与普通文件进行比较。

如果 file1 和 file2 都是目录，则 diff 会产生很多信息。如果一个目录中只有一个文件，则产生一条信息，指出该目录路径名和其中的文件名。

diff 命令的选项说明如表 1-7 所示。

表 1-7 diff 命令选项说明

选　项	说　明
-b	忽略行尾的空格，而字符串中的一个或多个空格符都视为相等
-c	采用上下文输出格式(提供三行上下文)
-C n	采用上下文输出格式(提供 n 行上下文)
-e	产生一个合法的 ed 脚本作为输出
-r	当 file1 和 file2 是目录时，递归作用到各文件和目录上

1.3.3.7 文件复制命令

Linux 下的 cp 命令用于复制文件或目录，该命令是最重要的文件操作命令，该命令的使用方式如下：

```
cp [选项] 源文件或目录 目标文件或目录
```

该命令把指定的源文件复制到目标文件或把多个源文件复制到目标目录中。

cp 命令的选项说明如表 1-8 所示。

表 1-8 cp 命令选项说明

选　项	说　明
-a	该选项通常在复制目录时使用。它保留链接、文件属性，并递归地复制目录，其作用等于 dpr 选项的组合
-d	复制时保留链接
-f	删除已经存在的目标文件而不提示
-i	和 f 选项相反，在覆盖目标文件之前将给出提示要求用户确认。回答 y 时目标文件将被覆盖，是交互式复制
-p	此时 cp 除复制源文件的内容外，还将把其修改时间和访问权限也复制到新文件中
-r	若给出的源文件是一目录文件，此时 cp 将递归复制该目录下所有的子目录和文件。此时目标文件必须为一个目录名
-l	不作复制，只是链接文件

需要说明的是，为防止用户在不经意的情况下用 cp 命令破坏另一个文件，如果指定的目标文件名是一个已存在的文件名，用 cp 命令复制文件后，这个文件就会被新复制的源文件覆盖，因此，在使用 cp 命令复制文件时，最好使用-i 选项。

例 1-10 将/home/lxy/test 目录下的文件复制到/home/lxy/test1 目录下，命令及响应如下：

```
root@lxy-desktop:/home/lxy/test1# ls ../test
2.6.22-vmhgfs-55017.tar.bz2    file4.txt~
file1.txt                      text
file1.txt~                     viewarticle.php.html
file2.txt                      vmhgfs.tar
file2.txt~                     vmxnet-2.6.22-rc1-vmws6(2).tar
file3.txt                      vmxnet.tar
file4.txt
root@lxy-desktop:/home/lxy/test1# ls
file1.txt   file1.txt~   file2.txt   file2.txt~   file3.txt   file4.txt   file4.txt~
root@lxy-desktop:/home/lxy/test1# cp ../test/text ./
root@lxy-desktop:/home/lxy/test1# ls
file1.txt     file2.txt     file3.txt    file4.txt~
file1.txt~    file2.txt~    file4.txt    text
```

可以看到，命令执行后，text 已经从/home/lxy/test 目录中复制到了/home/lxy/test1 目录中。

1.3.3.8 移动文件

在 Linux 系统中，移动文件可使用 mv 命令。mv 命令还可更改文件名，即把源文件以一个新文件名移动到另一个新的目录中去。该命令的使用方式如下：

```
mv [选项] 源文件名 目标文件名
mv [选项] 源目录名 目标目录名 2
mv [选项] 文件列表 目录
```

mv 命令的选项说明如表 1-9 所示。

表 1-9 mv 命令选项说明

选　　项	说　　明
-b	当遇到要覆盖其他文件或目录时，将自动备份，备份文件名为原文件名加上 -S 参数指定的字符串，若未设置则加上"~"
-i	交互模式，当移动的目录已存在同名的目标文件名时，用覆盖方式写文件，但在写入之前给出提示
-f	通常情况下，目标文件存在但用户没有写权限时，mv 会给出提示。本选项会使 mv 命令执行移动而不给出提示
-u	当要覆盖的文件或目录比源文件要新，则不覆盖目标文件
-S <字符串>	指定备份文件名后要加上的字符串

例 1-11 将 text 文件改名为 text1，命令及响应如下所示：

```
root@lxy-desktop:/home/lxy/test1# ls
text
root@lxy-desktop:/home/lxy/test1# mv text text1
root@lxy-desktop:/home/lxy/test1# ls
text1
```

从显示结果可以看出，执行 mv 命令后，text 文件已经成功地改名为 text1。

例 1-12 将/home/lxy/test1 目录中的所有文件移至/home/lxy/test 目录中，命令及响应如下：

```
root@lxy-desktop:/home/lxy/test1# ls
file1.txt    file2.txt    file3.txt   file4.txt~   text1
file1.txt~   file2.txt~   file4.txt   text
root@lxy-desktop:/home/lxy/test1# mv * ../test
root@lxy-desktop:/home/lxy/test1# ls
root@lxy-desktop:/home/lxy/test1# ls ../test
file1.txt    file2.txt    file3.txt   file4.txt~   text1
file1.txt~   file2.txt~   file4.txt   text
```

1.3.3.9 文件内容统计命令

wc 命令的功能为统计指定文件中的字节数、字数、行数，并将统计结果显示输出。该命令使用方式如下：

```
wc  [选项] 文件列表
```

该命令统计给定文件中的字节数、字数、行数。如果没有给出文件名，则从标准输入读取。wc 同时也给出所有指定文件的总统计数。字是由空格字符区分开的最大字符串。

wc 命令的选项说明如表 1-10 所示。

表 1-10　wc 命令选项说明

选　　项	说　　明
-c	统计字节数
-l	统计行数
-w	统计字数

例 1-13　统计 text 文件中的字节数。命令及响应如下：

```
root@lxy-desktop:/home/lxy/test# wc -c text
57 text
```

从显示结果可以看出，text 文件共包含 57 个字节。

1.3.4　目录及其操作命令

Linux 系统以文件目录的方式来组织和管理系统中的所有文件。所谓文件目录就是将所有文件的说明信息采用树型结构组织起来，即常说的目录。也就是说，整个文件系统有一个“根”(/)，然后在根上分“杈”(directory)，任何一个分杈上都可以再分杈，杈上也可以长出“叶子”。“根”和“杈”在 Linux 中被称为是“目录”或“文件夹”。而“叶子”则是一个个的文件。实践证明，此种结构的文件系统效率比较高。本节介绍一下 Linux 下的目录结构以及常用的目录操作等。

1.3.4.1　树型目录结构

Linux 目录结构如图 1-13 所示。

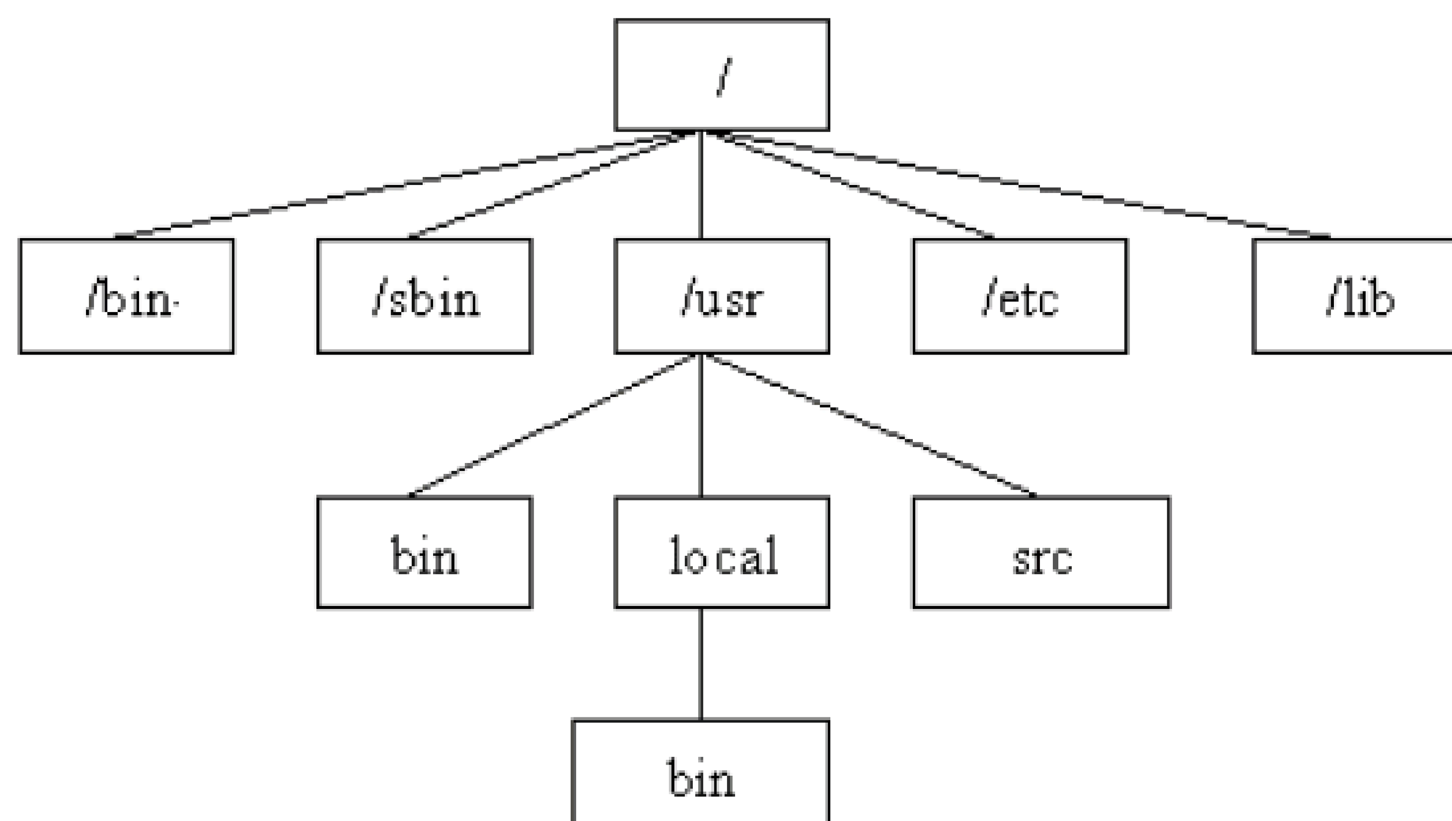


图 1-13　Linux 目录结构

如前所述，目录也是一种类型的文件。Linux 系统通过目录将系统中所有的文件分级、分层组织在一起，形成了 Linux 文件系统的树型层次结构。以根目录为起点，所有其他的目录都由根目录派生而来，用户可以浏览整个系统，可以进入任何一个已授权进入的目录，访问那里的文件。

Linux 目录提供了管理文件的一个方便途径。每个目录里面都包含文件。用户可以为特定的文件创建特定的目录，也可以把一个目录下的文件移动或复制到另一目录下，而且能移动整个目录，并且和系统中的其他用户共享目录和文件。

需要说明的是，根目录(系统目录)是 Linux 系统中的特殊目录。Linux 是一个多用户系统，操作系统本身的驻留程序存放在以根目录开始的专用目录中。

1.3.4.2 工作目录、用户主目录与路径

从逻辑上讲，用户在登录到 Linux 系统中之后，每时每刻都“处在”某个目录之中，此目录被称作工作目录或当前目录(Working Directory)。工作目录是可以随时改变的。用户初始登录到系统中时，其主目录(Home Directory)就成为其工作目录。工作目录用“.”表示，其父目录用“..”表示。

用户主目录是系统管理员增加用户时建立起来的(以后也可以改变)，每个用户都有自己的主目录，不同用户的主目录一般互不相同。 用户刚登录到系统中时，其工作目录便是该用户主目录，通常与用户的登录名相同。

路径是指从树型目录中的某个目录层次到某个文件的一条道路。此路径的主要构成是目录名称，中间用“/”分开。某个文件在文件系统中的位置都是由相应的路径决定的。

路径又分相对路径和绝对路径。 绝对路径是指从“根”开始的路径，也称为完全路径；相对路径是从用户工作目录开始的路径。应该注意到，在树型目录结构中到某一确定文件的绝对路径和相对路径均只有一条。绝对路径是确定不变的，而相对路径则随着用户工作目录的变化而不断变化。

1.3.4.3 Linux 系统主要目录说明

安装完Linux系统之后，在根文件下有许许多多的目录，无论哪个版本的 Linux 系统，都有这些目录，这些目录应该是标准的。当然各个 Linux 发行版本也会存在一些小小的差异，但总体来说，还是大体差不多。下面就简要介绍一下这些系统目录。

(1) /bin：该目录中存放 Linux 的常用命令，在有的版本中是一些和根目录下相同的目录。

(2) /boot：在这个目录下存放的都是系统启动时要用到的程序。在使用 lilo 引导 Linux 的时候，会用到这里的一些信息。

(3) /dev：该目录包含了 Linux 系统中使用的所有外部设备，它实际上是访问这些外部设备的端口，可以访问这些外部设备，与访问一个文件或一个目录没有区别。例如在系统中键入 cd /dev/cdrom，就可以看到光驱中的文件；键入 cd /dev/mouse 即可查看与鼠标相关的文件。

(4) /etc：etc 这个目录 Linux 系统中最重要的目录之一。该目录存放了系统管理时要用到的各种配置文件和子目录，例如网络配置文件、文件系统、X 系统配置文件、设备配置信息、设置用户信息等。

(5) /sbin：这个目录用来存放系统管理员的系统管理程序，是超级权限用户 root 的可执行命令存放地，普通用户无权限执行这个目录下的命令，这个目录和/usr/sbin；/usr/X11R6/sbin 或/usr/local/sbin 目录是相似的，凡是目录 sbin 中包含的都是 root 权限才能执行的命令。

(6) /home：如果建立一个用户，用户名是 lxy,那么在/home 目录下就有一个对应的/home/lxy 路径，用来存放用户的主目录。

(7) /lib：lib 是库(library)英文缩写。这个目录是用来存放系统动态连接共享库的。几乎所有的应用程序都会用到这个目录下的共享库。

(8) /lost+found：该目录在大多数情况下都是空的。但当突然停电或者非正常关机后，在重新启动机器的时候，有些文件就会找不到应该存放的地方，对于这些文件，系统将它们放在这个目录下。

(9) /mnt：这个目录在一般情况下也是空的。可以临时将别的文件系统挂在这个目录下。

(10) /media：即插即用型存储设备的挂载点自动在这个目录下创建，比如 USB 盘系统自动挂载后，会在这个目录下产生一个目录；CDROM/DVD 自动挂载后，也会在这个目录中创建一个目录，类似 cdrom 的目录。可以参看/etc/fstab 的定义。

(11) /opt：表示的是可选择的意思，有些软件包也会被安装在这里，也就是自定义软件包。

(12) /proc：操作系统运行时，进程(正在运行中的程序)信息及内核信息(比如 CPU、硬盘分区、内存信息等)存放在这里。可以在该目录下获取系统信息，这些信息是在内存中由系统自己产生的。

(13) /tmp：临时文件目录，有时用户运行程序的时候，会产生临时文件。/tmp 用来存放这些临时文件。

(14) /usr：是系统存放程序的目录，比如命令、帮助文件等，这是 Linux 系统中占用硬盘空间最大的目录。这个目录下有很多的文件和目录。当安装一个 Linux 发行版官方提供的软件包时，大多安装在这里。如果有涉及服务器配置文件的，会把配置文件安装在/etc 目录中。/usr 目录下包括涉及字体目录/usr/share/fonts，帮助目录 /usr/share/man 或/usr/share/doc，普通用户可执行文件目录/usr/bin 或/usr/local/bin 或/usr/X11R6/bin ，超级权限用户 root 的可执行命令存放目录，比如 /usr/sbin 或/usr/X11R6/sbin 或/usr/local/sbin 等；还有程序的头文件存放目录/usr/include。

(15) /var：这个目录的内容是经常变动的，/var 下有/var/log，这是用来存放系统日志的目录。/var/lib 用来存放一些库文件，比如 MySQL 的库文件，以及 MySQL 数据库的存放地等。

1.3.4.4 目录操作命令

为使读者更好地使用目录，本节将介绍一些常见的目录操作。

1. 创建目录

在 Linux 系统中建立新目录的命令是 mkdir。该命令的使用方式如下：

```
mkdir  [选项] 目录
```

mkdir 命令的选项说明如表 1-11 所示。

表 1-11 mkdir 命令选项说明

选项	说明
-m	在建立目录时按模式指定设置目录权限。该目录的权限分为：目录所有者的权限、组中其他人对目录的权限和系统中其他人对目录的权限。这三个权限分别用三个数字之和来表示：对目录的读权限是 4、写权限是 2、执行权限是 1
-p	可以是一个路径名称。此时若路径中的某些目录尚不存在，加上此选项后，系统将自动建立好那些尚不存在的目录，即一次可以建立多个目录

例 1-14 在当前目录中创建嵌套的目录层次 test 和 test 下的 mail 目录，权限设置为只有文件所有者有读、写和执行权限。命令及响应如下：

```
root@lxy-desktop:/home/lxy# mkdir -p -m 700 ./test/mail
root@lxy-desktop:/home/lxy# cd test
root@lxy-desktop:/home/lxy/test# ls
mail
```

从显示结果可以看出，执行 mkdir 命令后，成功地在 test 目录下建立了 mail 目录。

2. 删除目录

与创建目录对应的是删除目录，rmdir 命令用来删除目录，一般情况下要删除的目录必须为空目录，如果所给的目录不为空，系统会报告错误。该命令的使用方式如下：

```
rmdir [选项] 目录列表
```

rmdir 命令的选项说明如表 1-12 所示。

表 1-12 rmdir 命令选项说明

选项	说明
-p	在删除目录表指定的目录后，若父目录为空，则 rmdir 也删除父目录。状态信息显示什么被删除，什么没被删除

例 1-15 在当前目录下的 test 目录中，删除名为 mail 的子目录。若 mail 删除后，test 目录成为空目录，则 test 也会被删除。命令及响应如下：

```
root@lxy-desktop:/home/lxy/lxy# ls
test    test1
root@lxy-desktop:/home/lxy/lxy# ls test
mail
root@lxy-desktop:/home/lxy/lxy# rmdir -p test/mail
root@lxy-desktop:/home/lxy/lxy# ls
test1
```

从显示结果可以看出，执行删除命令后，当前目录中只剩下 test1 一个子目录。

3. 显示当前目录

显示当前目录的命令是 pwd 命令，该命令的使用方式如下：

```
pwd
```

例 1-16 显示当前目录，命令及响应如下：

```
root@lxy-desktop:/home/lxy# pwd
/home/lxy
```

从显示结果可以看出，当前目录是/home/lxy

4. 改变当前工作目录

改变当前工作目录在 Linux 系统中使用的是 cd 命令。该命令的使用方式如下：

```
cd     [directory]
```

该命令将当前目录改变至 directory 所指定的目录。若没有指定 directory，则回到用户的主目录。为了改变到指定目录，用户必须拥有对指定目录的执行和读权限。该命令可以使用通配符。

例 1-17 假设用户当前目录是：/home/lxy，现需要更换到/home/lxy/test 目录中，命令及响应如下：

```
root@lxy-desktop:/home/lxy# cd test
root@lxy-desktop:/home/lxy/test# pwd
/home/lxy/test
```

从显示结果可以看出，执行完 cd 命令后，当前目录已经变为/home/lxy/test。

例 1-18 从/home/lxy/test 返回到上一级目录，命令及响应如下：

```
root@lxy-desktop:/home/lxy/test# cd ..
root@lxy-desktop:/home/lxy# pwd
/home/lxy/
```

从显示结果可以看出，执行完命令后，成功地返回到/home/lxy 目录。

5. 链接文件的命令

链接文件的命令是 ln 命令。该命令在文件之间创建链接。这种操作实际上是给系统中已有的某个文件指定另外一个可用于访问它的名称。对于这个新的文件名，我们可以为之指定不同的访问权限，以控制对信息的共享和安全性的问题。如果链接指向目录，用户就可以利用该链接直接进入被链接的目录而不用输入一大堆的路径名。而且，即使删除这个链接，也不会破坏原来的目录。该命令的使用方式如下：

```
ln [选项] 源文件或目录 [链接名]
```

ln 命令的选项说明如表 1-13 所示。

表 1-13　ln 命令选项说明

选　项	说　明
-b	将在链接时会被覆盖或删除的文件进行备份
-d	允许系统管理员硬链接自己的目录
-f	链接时先将与链接名同名的文件删除
-i	在删除与链接名同名的文件时先进行询问
-s	进行符号链接
-v	在链接之前显示其文件名
-S SUFFIX	将备份的文件都加上 SUFFIX 的字尾
-V MTHOD	-V METHOD：指定备份的方式

链接有两种，一种被称为硬链接(Hard Link)，另一种被称为符号链接(Symbolic Link)。硬链接的意思是一个档案可以有多个名称，而符号链接的方式则是产生一个特殊的文件，该文件的内容是指向另一个文件的位置。建立硬链接时，链接文件和被链接文件必须位于同一个文件系统中，并且不能建立指向目录的硬链接。而对符号链接，则不存在这个问题。默认情况下，ln 产生硬链接。

例 1-19　为当前目录下的 text 文件建立一个名为 111 的符号链接，命令及响应如下：

```
root@lxy-desktop:/home/lxy/test# ln -s text 111
root@lxy-desktop:/home/lxy/test# ls
111    text
root@lxy-desktop:/home/lxy/test# cat text
sdfsfd
dsfsfdsdf
root@lxy-desktop:/home/lxy/test# cat 111
sdfsfd
dsfsfdsdf
```

可以看到，111 的文件内容同 text 的文件内容完全一致。

6. 改变文件或目录权限

在 Linux 系统中，用户设定文件权限控制其他用户不能访问、修改。但在系统应用中，有时需要让其他用户使用某个原来其不能访问的文件或目录，这时就需要重新设置文件的权限，使用的是 chmod 命令。并不是任何人都可改变文件和目录的访问权限，只有文件和目录的所有者才有权修改其权限，另外超级用户可对所有文件或目录进行权限设置。该命令的使用方式如下：

```
chmod [who] [+ | - | =] [mode]  文件名
```

chmod 命令中的操作对象 who 可以是表 1-14 字母中的任一个或者它们的组合。

表 1-14　chmod 命令 who 选项说明

选　项	说　明
u	表示“用户(user)”，即文件或目录的所有者
g	表示“同组(group)用户”，即与文件属主有相同组 ID 的所有用户
o	表示“其他(others)用户”
a	表示“所有(all)用户”。它是系统默认值

操作符号说明如表 1-15 所示。

表 1-15　chmod 命令操作符号说明

选　项	说　明
+	添加某个权限
-	取消某个权限
=	赋予给定权限并取消其他所有权限(如果有的话)

mode 所表示的权限可以是表 1-16 中字母的任意组合。

表 1-16　chmod 命令 mode 选项说明

选　项	说　明
r	可读
w	可写
x	可执行
X	只有目标文件对某些用户是可执行的或该目标文件是目录时才追加 x 属性
s	在文件执行时把进程的属主或组 ID 置为该文件的文件属主。方式“u+s”设置文件的用户 ID 位，“g+s”设置组 ID 位
t	保存程序的文本到交换设备上
u	与文件属主拥有一样的权限
g	与和文件属主同组的用户拥有一样的权限
o	与其他用户拥有一样的权限

在一个命令行中可给出多个权限方式，其间用逗号隔开。例如：chmod g+r，o+r example，这个命令将使同组和其他用户对文件 example 有读权限。

文件和目录的权限还可用八进制数字模式来表示。首先了解用数字表示的属性的含义：0 表示没有权限，1 表示可执行权限，2 表示可写权限，4 表示可读权限，然后将其相加。所以数字属性的格式应为 3 个从 0 到 7 的八进制数，其顺序是(u)(g)(o)。例如，如果想让某个文件的属主有“读/写”两种权限，需要把 4(可读)+2(可写)＝6(读/写)。

数字设定法的一般形式为：

```
chmod  [mod]  文件名
```

例 1-20 设定文件 text 的属性为：文件属主(u)增加执行权限，与文件属主同组用户(g)增加执行权限，其他用户(o)增加执行权限。命令及响应如下：

```
root@lxy-desktop:/home/lxy/test# ls -l text
-rw-r--r-- 1 root root 17 Dec 24 20:00 text
root@lxy-desktop:/home/lxy/test# chmod a+x text
root@lxy-desktop:/home/lxy/test# ls -l text
-rwxr-xr-x 1 root root 17 Dec 24 20:00 text
```

从显示结果可以看出，chmod 命令执行前，text 文件属性为-rw-r--r--，chmod 命令执行后，文件属性变为-rwxr-xr-x。

例 1-21 设定文件 text 的属性为：文件属主(u)增加写权限，与文件属主同组用户(g) 增加写权限，其他用户(o)删除执行权限。命令及响应如下：

```
root@lxy-desktop:/home/lxy/test# ls -l text
-rwxr-xr-x 1 root root 0 Dec 24 20:43 text
root@lxy-desktop:/home/lxy/test# chmod ug+w,o-x text
root@lxy-desktop:/home/lxy/test# ls -l text
-rwxrwxr-- 1 root root 0 Dec 24 20:43 text
```

从显示结果可以看出，text 文件属性已经从-rwxr-xr-x 变为-rwxrwxr--。

7. 改变文件或目录的属主和属组

chown 命令用来更改某个文件或目录的属主和属组。这个命令也很常用。例如 root 用户把自己的一个文件复制给用户 lxy，为了让用户 lxy 能够存取这个文件，root 用户应该把这个文件的属主设为 lxy，否则，用户 lxy 无法存取这个文件。该命令的使用方式如下：

```
chown [选项] 用户或组  文件
```

chown 将指定文件的拥有者改为指定的用户或组。用户可以是用户名或用户 ID。组可以是组名或组 ID。文件是以空格分开的要改变权限的文件列表，支持通配符。chown 命令的选项说明如表 1-17 所示。

表 1-17 chown 命令选项说明

选 项	说 明
-c	若该文件拥有者确实已经更改，才显示其更改动作
-f	若该文件拥有者无法被更改也不要显示错误信息
-i	在删除与链接名同名的文件时先进行询问
-h	只对链接进行变更，而非该链接真正指向的文件
-v	显示拥有者变更的详细资料
-R	递归式地改变指定目录及其下的所有子目录和文件的拥有者

例 1-22 把当前目录下的 text 文件所有者改为 lxy。命令及响应如下：

```
root@lxy-desktop:/home/lxy/test# ls -l text
-rw-r--r-- 1 root root 0 Dec 24 20:57 text
root@lxy-desktop:/home/lxy/test# chown lxy text
root@lxy-desktop:/home/lxy/test# ls -l text
-rw-r--r-- 1 lxy root 0 Dec 24 20:57 text
```

从显示结果可以看出，text 文件在命令执行后所有者已经从 root 变更为 lxy。

例 1-23 把目录 test 及其下的所有文件和子目录的属主改成 lxy，属组改成 users。命令及响应如下：

```
root@lxy-desktop:/home/lxy# ls -l test
total 0
-rw-r--r-- 1 root root 0 Dec 24 20:57 text
root@lxy-desktop:/home/lxy# chown -R lxy.users test
root@lxy-desktop:/home/lxy# ls -l test
total 0
-rw-r--r-- 1 lxy users 0 Dec 24 20:57 text
```

从显示结果可以看出，test 目录在命令执行后属主已经从 root 变更为 lxy，属组从 root 变更为 users。

1.3.5 文件压缩命令

压缩文件大小有两个明显的好处，一是可以减少存储空间，二是通过网络传输文件时，可以减少传输的时间。Linux 提供了很多与文件打包、压缩有关的命令。下面介绍一些常用的文件打包压缩命令。

1.3.5.1 文件压缩

gzip 是在 Linux 系统中经常使用的一个对文件进行压缩和解压缩的命令，既方便又好

用。该命令的使用方式如下：

```
gzip [选项] 压缩(解压缩)的文件名
```

gzip 命令的选项说明如表 1-18 所示。

表 1-18 gzip 命令选项说明

选　项	说　明
-c	将输出写到标准输出上，并保留原有文件
-d	将压缩文件解压
-l	对每个压缩文件，显示下列字段：压缩文件的大小；未压缩文件的大小；压缩比；未压缩文件的名字
-r	递归式地查找指定目录并压缩其中的所有文件或者是解压缩
-t	测试、检查压缩文件是否完整
-v	对每一个压缩和解压的文件，显示文件名和压缩比
-num	用指定的数字 num 调整压缩的速度，-1 或 --fast 表示最快压缩方法(低压缩比)，-9 或--best 表示最慢压缩方法(高压缩比)。系统默认值为 6

例 1-24 把当前目录下的每个文件压缩成 .gz 文件，命令及响应如下：

```
root@lxy-desktop:/home/lxy/test# ls
text    text1    text~
root@lxy-desktop:/home/lxy/test# gzip *
root@lxy-desktop:/home/lxy/test# ls
text.gz    text1.gz    text~.gz
```

从显示结果可以看出，gzip 已经把 3 个文件进行了压缩，并且文件名后缀都改为 gz。

例 1-25 把当前目录下每个压缩的文件解压，并列出详细的信息。

```
root@lxy-desktop:/home/lxy/test# gzip -dv *
text.gz:          61.3% -- replaced with text
text1.gz:         43.2% -- replaced with text1
text~.gz:          0.0% -- replaced with text~
root@lxy-desktop:/home/lxy/test# ls
text    text1    text~
```

例 1-26 是例 1-24 的反过程，从显示结果可以看出，gzip 又把压缩文件进行了解压，恢复成了原始文件。

1.3.5.2 文件打包

利用 tar 命令，可以把一大堆的文件和目录全部打包成一个文件，这对于备份文件或将几个文件组合成为一个文件以便于网络传输是非常有用的。tar 可以为文件和目录创建备

份。利用 tar，用户可以为某一特定文件创建备份，也可以在备份中改变文件，或者向备份中加入新的文件。tar 最初被用来在磁带上创建备份，现在，用户可以在任何设备上创建备份。该命令的使用方式如下：

```
tar [主选项+辅选项] 文件或者目录
```

tar 命令有主选项和辅选项，主选项是必须要有的，它告诉 tar 要做什么事情，辅选项是辅助使用的，可以选用。

tar 命令的主选项和辅选项说明分别如表 1-19 和表 1-20 所示。

表 1-19 tar 命令主选项说明

选　项	说　明
-c	创建新的备份文件
-r	把要存档的文件追加到备份文件的未尾
-t	列出备份文件的内容，查看已经备份了哪些文件
-u	更新文件，用新增的文件取代原备份文件，如果在备份文件中找不到要更新的文件，则把它追加到备份文件的最后
-x	从备份文件中释放文件

表 1-20 tar 命令辅选项说明

选　项	说　明
-b	该选项是为磁带机设定的。其后跟一数字，用来说明区块的大小，系统预设值为 20 (20*512 bytes)
-f	使用备份文件或设备，这个选项通常是必选的
-k	保存已经存在的文件。例如我们把某个文件还原，在还原的过程中，遇到相同的文件，不会进行覆盖
-m	在还原文件时，把所有文件的修改时间设定为现在
-M	创建多卷的备份文件，以便在几个磁盘中存放
-v	详细报告 tar 处理的文件信息。如无此选项，tar 不报告文件信息
-w	每一步都要求确认
-z	用 gzip 来压缩/解压缩文件，加上该选项后可以将档案文件进行压缩，但还原时也一定要使用该选项进行解压缩

例 1-26 把 test 目录下包括它的子目录全部做备份文件，备份文件名为 test.tar，命令及响应如下：

```
root@lxy-desktop:~/lxy# ls
test    test1
```

```
root@lxy-desktop:~/lxy# ls ./test
text1    text2    text3
root@lxy-desktop:~/lxy# tar -cvf test.tar ./test
./test/
./test/text3
./test/text2
./test/text1
root@lxy-desktop:~/lxy# ls
test    test1    test.tar
```

第 1 个命令是显示当前目录下的内容，可以看到当前目录包含 test 子目录，第 2 个命令是显示 test 子目录的内容，可以看到 test 目录下有 3 个文件，执行 tar 命令后，再执行 ls 命令可以看到当前目录下多了一个 test.tar 文件。

1.3.6 联机帮助命令

Linux 提供的命令很多，在学习 Linux 的过程中，不可避免地要遇到一些不会使用的命令，这时可以通过 Linux 提供的一些帮助命令来了解命令的使用方法。学会使用这些帮助，对于快速掌握 Linux 是必不可少的。下面介绍一下常用的帮助命令。

1.3.6.1 显示帮助手册

在 Linux 下想获得一个命令的帮助，可以使用 man 命令。只要在命令 man 后输入想要获取的命令的名称(例如 ls)，man 就会列出一份完整的说明，其内容包括命令语法、各选项的意义以及相关命令等。该命令使用方式如下：

```
man    [选项]   命令名称
```

man 命令的常用选项说明如表 1-21 所示。

表 1-21　man 命令选项说明

选　项	说　明
-f	只显示出命令的功能而不显示其中详细的说明文件
-w	不显示手册页，只显示将被格式化和显示的文件所在位置
-a	显示所有的手册页，而不是只显示第一个
-E	在每行的末尾显示$符号

1.3.6.2 查看命令帮助

help 命令用于查看所有 Shell 命令。用户可以通过该命令寻求 Shell 命令的用法，只需在所查找的命令后输入 help 命令，就可以看到所查命令的内容了。例如：输入 cd –help 便

可查看 cd 命令的使用方法。

info 命令用来获取相关命令的详细使用方法，例如：info ls 可以获取如何使用 info 的详细信息。

1.3.6.3 查找文件在文件系统中的位置

whereis 命令用来定位可执行文件、源代码文件、帮助文件在文件系统中的位置。例如，最常用的 ls 命令，它是在/bin 这个目录下的。如果希望知道某个命令存在哪一个目录下，可以用 whereis 命令来查询。该命令的使用方式如下：

```
whereis  [选项]  命令名
```

whereis 命令的常用选项说明如表 1-22 所示。

表 1-22 whereis 命令选项说明

选 项	说 明
-b	定位可执行文件
-m	定位帮助文件
-s	定位源代码文件
-u	搜索默认路径下除可执行文件、源代码文件、帮助文件以外的其他文件
-B	指定搜索可执行文件的路径
-M	指定搜索帮助文件的路径

例 1-27 查找 ls 命令的二进制文件在什么目录下，命令及响应如下：

```
root@lxy-desktop:~# whereis -b ls
ls: /bin/ls
```

从显示结果可以看出，ls 命令位于/bin/目录下。

1.3.7 用户操作命令

Linux 是一种多用户操作系统，如果所有用户共享一个账号，会造成许多麻烦。因此在 Linux 中每个用户都有自己的账号。各个用户的账号可以根据需要分配不同的权限。Linux 提供了与之相关的用户操作命令。

1.3.7.1 切换用户命令

su 命令用来切换用户身份，该命令的使用方式如下：

```
su  [选项]  user
```

除 root 外，其他用户切换身份时，需输入密码。su 命令的常用选项说明如表 1-23 所示。

表 1-23　su 命令选项说明

选　项	说　明
-p	执行 su 时不改变环境参数
-c	切换到 user 用户并执行指令(command)，然后再切换回原来用户
-s	shell 指定要执行的 shell，默认在/etc/passwd 文件中已设置完成，若用户需改 Shell 时，可采用此参数

1.3.7.2　sudo 命令

sudo 命令用来以系统管理员的身份执行指令，该命令的使用方式如下：

```
sudo [选项] 命令
```

以系统管理者的身份执行指令，也就是说，经由 sudo 所执行的指令就好像是 root 亲自执行。sudo 命令的常用选项说明如表 1-24 所示。

表 1-24　sudo 命令选项说明

选　项	说　明
-l	显示出执行 sudo 的用户的权限
-v	sudo 在第一次执行时或是在 N 分钟内没有执行(N 预设为 5)会问密码，这个参数是重新做一次确认，如果超过 N 分钟，也会问密码
-k	强迫用户在下一次执行 sudo 时问密码(不论有没有超过 N 分钟)
-b	执行的指令放在后台执行
-p prompt	更改问密码的提示语，其中 %u 会替换为使用者的账号名称， %h 会显示主机名称
-u username/#uid	不加此参数，代表要以 root 的身份执行指令，而加了此参数，可以以 username 的身份执行指令(#uid 为该 username 的使用者账号)
-s	执行环境变量中的 SHELL 所指定的 shell ，或是 /etc/passwd 里所指定的 shell
-H	将环境变量中的 home 目录指定为要变更身份的使用者 home 目录(如不加 –u 参数就是系统管理者 root)

例 1-28　从当前用户 lxy 以系统管理员身份执行指令，命令及响应如下：

```
lxy@lxy-desktop:~$ sudo -s
[sudo] password for lxy:
root@lxy-desktop:~#
```

命令的第一行是请求切换到系统管理员，随后系统要求输入密码，在输入密码之后，可以看到已经从用户 lxy 切换到 root。

1.3.8 关机和重启计算机命令

由于 Linux 是一种多用户、多任务操作系统，因此在切断计算机电源之前，必须先关闭 Linux 系统。决不能不执行关机进程就切断计算机电源，这样做会导致保存在内存缓冲区的磁盘数据来不及写回磁盘，从而破坏文件系统。本节介绍一下与关机和重启计算机有关的命令。

1.3.8.1 shutdown 命令

shutdown 命令可以安全地关闭或重启 Linux 系统，它在系统关闭之前给系统上的所有登录用户提示一条警告信息。该命令还允许用户指定一个时间参数，可以是一个精确的时间，也可以是从现在开始的一个时间段。精确时间的格式是 hh:mm，表示小时和分钟；时间段由“+”和分钟数表示。系统执行该命令后，会自动进行数据同步的工作。该命令使用方式如下：

```
shutdown [选项] [时间] [警告信息]
```

shutdown 命令的常用选项说明如表 1-25 所示。

表 1-25 shutdown 命令选项说明

选　项	说　明
-k	并不真正关机，而只是发出警告信息给所有用户
-r	关机后立即重新启动
-h	关机后不重新启动
-c	取消一个已经运行的shutdown

需要特别说明的是，该命令只能由超级用户使用。

例 1-29 系统在 5 分钟后关机，并告诉所有用户。命令及响应如下所示：

```
root@lxy-desktop:/# shutdown -h +5 "The system is going down in 5 min"

Broadcast message from lxy@lxy-desktop
        (/dev/pts/0) at 21:05 ...

The system is going down for halt in 5 minutes!
The system is going down in 5 min
```

命令的第一行是向所有用户广播一个消息："The system is going down in 5 min"，随后在第 3 行可以看到收到了系统发布的消息，最后一行是 shutdown 命令中输入的广播消息的内容。

1.3.8.2 halt 命令

halt 是最简单的关机命令，其实际上是调用 shutdown -h 命令。halt 执行时，杀死应用进程，文件系统写操作完成后就会停止内核。该命令的使用方式如下：

```
halt [选项]
```

halt 命令的常用选项说明如表 1-26 所示。

表 1-26 halt 命令选项说明

选 项	说 明
-n	在关机前不做将内存资料写回硬盘的动作
-w	并不会真正关机，只是把记录写到 /var/log/wtmp 文件里
-d	不把记录写到 /var/log/wtmp 档案里(-n 这个参数包含了-d)
-f	强迫关机，不调用 shutdown 这个指令
-i	在关机之前先把所有网络相关的装置停止
-p	当关机的时候，顺便做关闭电源(poweroff)的动作。取消一个已经运行的shutdown

需要特别说明的是，该命令只能由超级用户使用。

1.3.8.3 reboot 命令

reboot 命令用来重新启动计算机。该命令的使用方式如下：

```
reboot [选项]
```

reboot 命令的常用选项说明如表 1-27 所示。

表 1-27 reboot 命令选项说明

选 项	说 明
-n	在关机前不做将内存资料写回硬盘的动作
-w	并不会真正关机，只是把记录写到 /var/log/wtmp 文件里
-d	不把记录写到 /var/log/wtmp 档案里(-n 这个参数包含了-d)
-f	强迫关机，不调用 shutdown 这个指令
-i	在关机之前先把所有网络相关的装置停止

1.4 小　结

本章主要介绍了 Linux 的一些基础知识，先说明了 Linux 的起源、特点、版本以及它的发展前景等，然后以 Ubuntu Linux 发行版为例，介绍了 Linux 的安装。最后，介绍了 Linux 的常用命令与操作。关于 Linux 更详细的操作，读者可以查阅其他有关 Linux 操作的书籍。通过本章的学习，读者应对 Linux 系统有一个初步的认识，从而为今后的学习打下基础。

习　题

一、填空题

1. Linux 操作系统是______ 操作系统的一个克隆版本。
2. Linux 的命令运行环境是______，它是一种命令解释器，在用户和操作系统之间提供了一个交互接口。
3. Linux 系统中有三种基本的文件类型：______、______和______。
4. Linux 系统通过______将系统中所有的文件分级、分层组织在一起，形成了 Linux 文件系统的树型层次结构。
5. 在 Linux 系统中建立新目录的命令是______。

二、选择题

1. 在______目录下存放的都是系统启动时要用到的程序。在使用 lilo 引导 Linux 的时候，会用到这里的一些信息。

   (A) /boot　(B) /bin　(C) /dev　(D) /etc

2. Linux 的工作目录用______表示。

   (A) .　(B) ,　(C) *　(D) #

3. 利用______命令，可以把一大堆的文件和目录全部打包成一个文件，这对于备份文件或将几个文件组合成为一个文件以便于网络传输是非常有用的。

   (A) gzip　(B) tar　(C) cd　(D) ls

4. 在 Linux 下想获得一个命令的帮助，可以使用______命令。

   (A) cd　(B) ls　(C) man　(D) gzip

5. ______命令用来重新启动计算机。

   (A) shutdown　(B) halt　(C) reboot　(D) quit

三、上机题

1. 选择一种 Linux 发行版本，并将它安装到计算机上。

2. 列出安装的 Linux 版本根目录下的子目录，看是否与本书列的一致。

3. 练习使用 Linux 的文件和目录操作命令，如建立新目录，删除目录，复制文件，压缩文件等。

4. 上机练习 Linux 关机和重启计算机命令。

CHAPTER 2

Linux 下的 C 语言编程环境

Linux 作为一个操作系统，一项重要的功能就是要支持用户编程。传统的 Unix 下的程序开发语言是 C 语言，C 语言是一种平台适应性强、易于移植的语言。Linux 是用 C 语言写成的，反过来，Linux 又为 C 语言提供了很好的支持，C 语言编译工具 gcc、调试工具 gdb 属于最早开发出来的一批自由软件。因此 Linux 与 C 语言形成了完美的结合，为用户提供了一个强大的编程环境。本章先介绍一下 Linux 下的 C 语言编程环境。

2.1 Linux 编程简介

Linux 编程可分为 Shell 编程和高级语言编程。其中 Shell 编程常用的语言有 BASH、TCSH、GAWK、Perl、Tcl 和 Tk 等。高级语言包括 C、C++、Java 等。所有的 Linux 程序需要首先转化为低级机器语言即所谓的二进制代码以后，才能被操作系统执行。编程时，先用普通的编程语言生成一系列指令，这些指令可被翻译为适当的可执行应用程序的二进制代码。这个翻译过程可由解释器一步步来完成，或者也可以立即由编译器明确地完成。Shell 编程语言都利用它们各自的解释器。用文本编辑器编辑好源程序后，不需要编译和链接可以直接执行。编译语言则不同，它将源程序进行编译和链接，生成一个独立的二进制代码文件，然后才可以运行它。比如 C 语言源程序需要先经过 gcc 编译器编译生成可执行的应用程序文件，然后才可以运行它。

Shell 程序将一些 Linux 命令结合起来完成一项特定功能。BASH Shell 提供了许多编程工具，可用来生成 Shell 程序。进行 Shell 编程需要熟练掌握这些实用工具并根据需要选择合适的实现方法，读者可以在关于 Linux 的实用工具和 Shell 编程的书里查阅这方面的内容，而本书主要集中讲解 Linux 平台下进行 C 语言开发的知识。

2.2　Linux 下的 C 语言开发环境

用 C 语言进行软件开发的一般步骤是：用编辑器编写源代码，源代码编写完成之后，通过编译器编译链接，编译链接成功之后，系统会生成一个可执行文件，执行该可执行文件，程序就开始运行。如果程序运行结果与预期结果不符，需要利用调试工具对程序进行调试，找出其中的错误所在，然后对源程序进行修改、编译、运行、调试，直至得到正确的结果为止。此外，在软件规模较大时，应用程序可能包括几百个源文件，因此也需要有效的程序源文件维护工具。

Linux 为软件开发者提供了强大的 C 语言开发环境和丰富的开发维护工具，熟悉并掌握这些工具是进行 Linux 平台软件开发的必要条件。这些工具有：

- 编辑工具：利用 Linux 自带的编辑工具 vi 和 emacs，可以用来编辑 C 语言源程序。
- 编译工具：Linux 带有功能强大的符合 ANSIC 标准的编译系统 gcc，利用 gcc 可以编译 C/C++语言源程序。
- 调试工具：利用 Linux 自带的调试工具 gdb，可以调试 C 语言程序。
- 维护工具：make 程序可以对程序源文件进行有效的管理。

在下面几节里，将逐步讲解这些工具的使用方法。

2.3　编辑器的使用

Linux 系统提供了许多文本编辑程序，比较常用的有 vi 和 emacs 等。vi 和 emacs 都是全屏幕编辑器，功能强大。本节简单地介绍一下它们的使用方法。

2.3.1　vi 的使用

vi 是 visual interface 的简称，它可以执行输出、删除、查找、替换块操作等众多文本功能而且用户可以根据自己的需要对其进行定制。vi 的许多功能是使用命令模式完成的，且命令繁多，这使得在一开始学习使用的时候会有些麻烦。实际上，如果真正熟练掌握了 vi，就会发现它很方便快捷，可以大量减少键盘的敲击量。

进入 vi 的方法很简单，在终端 shell 提示符后输入下列命令即可：

```
vi [选项] 文件名
```

其中，选项和文件名都是可选的。进入 vi 后的界面如图 2-1 所示。



图 2-1 vi 使用界面

vi 有三种基本工作模式：命令模式(Command Mode)、输入模式(Insert Mode)和最后行命令模式(Last Line Mode)。

(1) 在命令模式下，命令是在当前光标处进行操作的，它的主要功能是控制光标的移动、删除字符、复制段落以及进入输入模式或最后行命令模式。

(2) 输入模式的功能是输入文本数据，在输入模式下，按 Esc 键即可返回到命令模式。

(3) 最后行命令模式是指在屏幕终端的最后一行的操作命令，它可以执行一些比较特殊的功能，比如保存文件、退出 vi、在文件中进行文本的查找、替换字符串等其他操作。

在进行文本的输入时使用输入模式，而在执行某些特殊功能时，使用命令模式或最后行命令模式。在命令模式输入“：”即可进入最后行命令模式，此时，vi 会在显示窗口的最后一行显示一个“：”作为最后行命令模式的提示符(如图 2-2 所示)，等待用户输入命令。



图 2-2 vi 的最后行命令模式

在刚进入 vi 时的默认状态并不是输入模式，而是命令模式。可以输入命令“i”、附加命令“a”、打开命令“o”、修改命令“c”、取代命令“r”或替换命令“s”进入输入模式。在该模式下，用户输入的任何字符都被 vi 当作文件内容保存起来，并将其显示在屏幕上。

vi 命令中的常用选项如表 2-1 所示。

表 2-1　vi 命令选项说明

选　　项	说　　明
+n	打开文件，并将光标置于第 n 行行首
+	打开文件，并将光标置于最后一行行首
+/string	打开文件，并将光标置于第一个与 string 匹配的字符串处
-r	在上次使用 vi 编辑时发生系统崩溃，恢复文件。

进入 vi 之后，首先进入命令模式。光标停在屏幕第一行第一列上，其余各行行首都有一个“~”符号，表示该行为空行。最后一行也称状态行，显示出当前正在编辑的文件名及其状态。如果在命令中输入了文件名，若文件在系统中已经存在，vi 会在屏幕上显示出该文件的内容，在状态行显示该文件的文件名、行数和字符数，如图 2-3 所示。如果该文件不存在，则 vi 会新建一个文件。



图 2-3　vi 状态行

在命令模式下，可以输入不同的命令，这些命令主要包括切换到输入模式或最后行命令模式命令、移动光标类命令、屏幕翻滚类命令、删除命令、修改命令、其他功能命令等。由于命令实在太多，笔者在此仅整理常用的命令。各命令说明如表 2-2~表 2-7 所示。

表 2-2　切换到 Insert Mode 命令

命　　令	说　　明
i	插入，从当前光标所在之处插入所输入的文字
I	插入，从当前行首所在之处插入所输入的文字
a	增加，从当前光标所在的下一个字开始输入文字
A	增加，从当前行尾所在的下一个字开始输入文字
o	在当前行之下插入新的一行，从行首开始输入文字
O	在当前行之上插入新的一行，从行首开始输入文字

表 2-3　移动光标类命令

命　　令	说　　明
h 或 Backspace	光标左移一个字符
l 或 Space	光标右移一个字符
k 或 Ctrl+p	光标上移一行
j 或 Ctrl+n	光标下移一行
Enter	光标下移一行
w 或 W	光标右移一个字至字首
b 或 B	光标左移一个字至字首
e 或 E	光标右移一个字至字尾
)	光标移至句尾
(	光标移至句首
}	光标移至段落开头
{	光标移至段落结尾
nG	光标移至第 n 行行首
n+	光标下移 n 行
n-	光标上移 n 行
n$	光标移至第 n 行尾
H	光标移至屏幕顶行
M	光标移至屏幕中间行
L	光标移至屏幕最后行
0	光标移至当前行首
$	光标移至当前行尾

表 2-4 屏幕翻滚类命令

命　令	说　明
Ctrl+u	向文件首翻半屏
Ctrl+d	向文件尾翻半屏
Ctrl+f	向文件尾翻一屏
Ctrl+b	向文件首翻一屏
nz	将第 n 行滚至屏幕顶部，不指定 n 时将当前行滚至屏幕顶部

表 2-5 删 除 命 令

命　令	说　明
ndw 或 ndW	删除光标处开始及其后的 n-1 个字
d0	删除从前一个字符到行首的所有字符
d$	删除从当前字符至行尾的所有字符
ndd	删除当前行及其后 n-1 行
x 或 X	删除一个字符，x(小写)删除光标后的，而 X(大写)删除光标前的
Ctrl+u	删除输入方式下所输入的文本

表 2-6 修 改 命 令

命　令	说　明
r	替换当前字符
R	替换当前字符及其后的字符，直至按 ESC 键
s	从当前光标位置处开始，以输入的文本替代指定数目的字符
S	删除指定数目的行，并以所输入的文本代替它
ncw 或 nCW	修改指定数目的字
nCC	修改指定数目的行

表 2-7 其他功能命令

命　令	说　明
u	取消前次操作
.	重复上次操作
/string	从光标开始处向文件尾搜索 string
?string	从光标开始处向文件首搜索 string
n	在同一方向重复上一次搜索命令
N	在反方向上重复上一次搜索命令
ZZ	保存并退出 vi，返回到 shell

在最后行命令模式下，vi 常用命令及命令说明如表 2-8 所示。

表 2-8 最后行命令模式命令

命 令	说 明
n1,n2 co n3	将 n1 行到 n2 行之间的内容复制到第 n3 行下
n1,n2 m n3	将 n1 行到 n2 行之间的内容移至第 n3 行下
n1,n2 d	将 n1 行到 n2 行之间的内容删除
e file	打开文件 file 进行编辑
x	保存当前文件并退出
q	退出 vi，返回到 shell
q!	不保存文件并退出 vi，直接返回到 shell
w	保存当前文件，但不退出 vi，而是继续等待用户输入命令
wq	先保存文件，然后退出 vi 返回到 shell
!command	执行 shell 命令 command
s/s1/s2/g	将当前行中所有 s1 均用 s2 替代
n1,n2s/s1/s2/g	将第 n1 至 n2 行中所有 s1 均用 s2 替代
g/s1/s//s2/g	将文件中所有 s1 均用 s2 替换

vi 是 shell 或终端下常用的文本编辑器，功能非常强大，但 vi 包含众多的命令，读者需要反复练习，才能最终达到熟练运用。本节只是简单介绍了 vi 的常用命令，关于 vi 详细使用方法，读者可查阅 vi 的帮助手册以及其他有关资料，在此不再赘述。

2.3.2 Emacs 的使用

Emacs 是 Linux 下另一个常见的功能强大的编辑器，它提供了对 Shell 和其他一些工具的支持。与 vi 一样，Emacs 很多功能也要通过快捷键来完成，因而熟练掌握 Emacs 需要记忆大量的命令。本节只介绍 Emacs 的简单应用，如何进行一般的编辑工作，不涉及复杂的 Emacs 技巧。

在 Ubuntu 下，Emacs 位于/usr/bin 目录下，在 X window 图形界面中，双击 Emacs 图标，或者在终端下输入：

```
emacs [选项] 文件名
```

即可进入 Emacs 编辑环境，其中选项和文件名是可选的。Emacs 分为图形界面的 xemacs 和文本界面的 Emacs。在 X window 和图形终端下启动的是 xemacs，在字符终端下启动的是文本界面的 Emacs。

进入 Emacs 后的界面如图 2-4 所示。

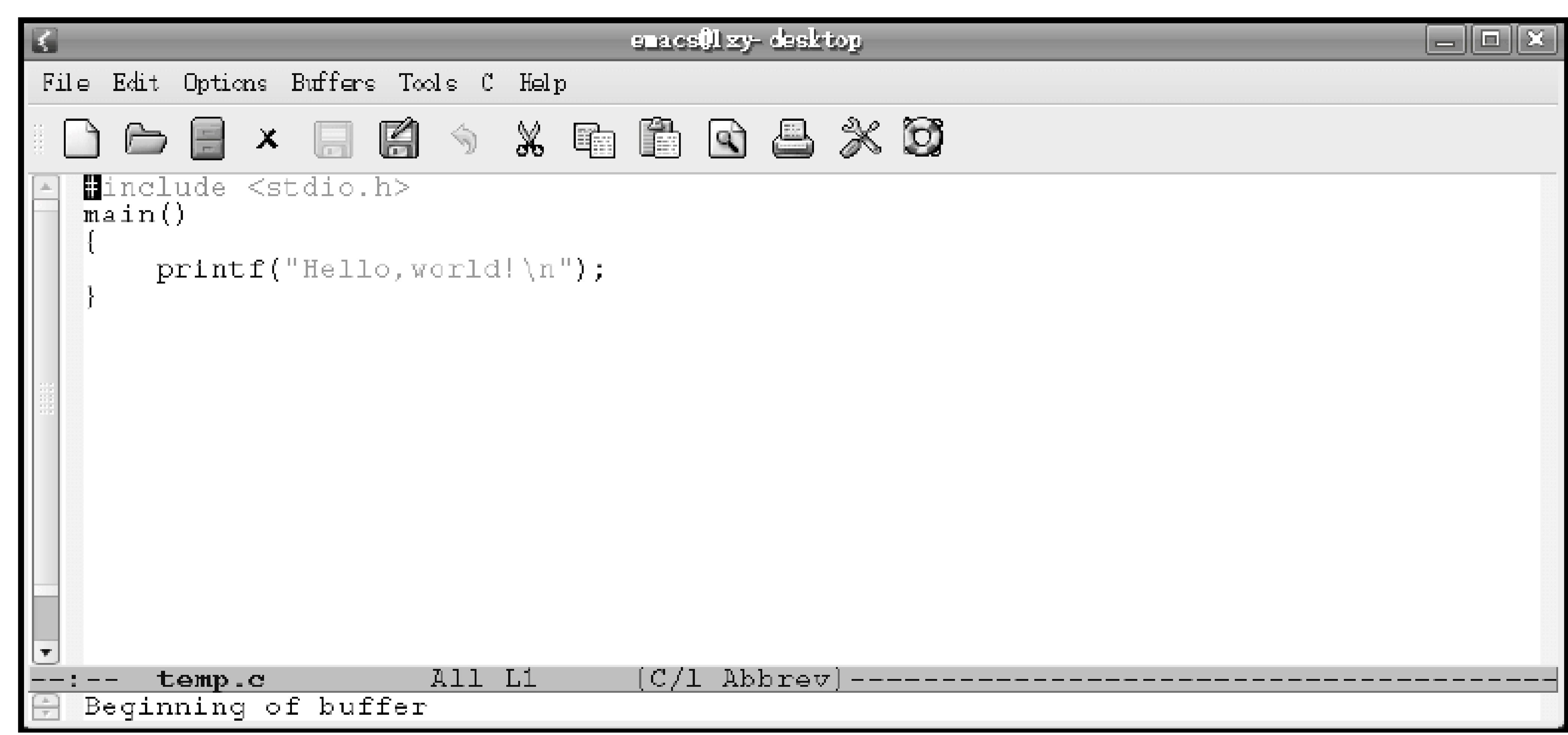


图 2-4 Emacs 使用界面

Emacs 命令中的常用选项如表 2-9 所示。

表 2-9 Emacs 命令选项说明

选 项	说 明
-nw	进入文本方式的 Emacs
+n	打开文件，并将光标置于第 n 行行首
+n:m	打开文件，并将光标置于第 n 行第 m 列

同 vi 一样，在输入命令时，如果输入了文件名，若该文件在系统中已经存在，Emacs 会在屏幕上显示出该文件的内容，如果该文件不存在，则 Emacs 会新建一个文件。Emacs 将所有被编辑的文件看成一张白纸，称为 buffer，编辑是在 buffer 上完成的。

按照 Emacs 的习惯，将键盘上的 Alt 键称为 Meta 键，以后将 Ctrl + x 之类的组合键记为 C-x,，Alt+x 记为 M-x，Emacs 的各种组合键除了一些标点以外都不区分大小写。Emacs 的很多功能都是通过组合键完成的，比如 C-x、C-f 表示连续输入 Ctrl+X 和 Ctrl+F。超过 2 个以上的按键命令，Emacs 会在屏幕最下面进行提示。

表 2-10 及表 2~13 列出了 Emacs 里的常用命令以及功能说明。

表 2-10 移动光标类命令

命 令	说 明
C-p	光标上移一行
C-n	光标下移一行
C-f	光标右移一个字符
M-f	光标右移一个字
C-b	光标左移一个字符
M-b	光标左移一个字
C-a	光标移至当前行首

(续表)

命　令	说　明
C-e	光标移至当前行尾
M-a	光标移至当前句首
M-e	光标移至当前句尾
M-<	光标移至文件头
M->	光标移至文件尾

表 2-11　屏幕翻滚类命令

命　令	说　明
C-v	向文件尾翻一屏
M-v	向文件首翻一屏

表 2-12　编辑命令

命　令	说　明
Backspace	删除光标前的一个字符
C-d	删除光标后的一个字符
M-Backspace	剪切光标前的一个词
M-d	剪切光标后的一个词
C-k	剪切从光标到行尾间的字符
M-k	剪切从光标到句尾间的字符
C-y	粘贴剪切的内容

表 2-13　其他功能命令

命　令	说　明
C-x C-f	打开文件
C-x C-s	保存文件
C-x s	提示保存所有文件
C-g	取消当前操作
C-x C-w	文件另存为
C-x　C-c	不保存文件退出 Emacs

前面只是列出了 Emacs 常用的一些命令，关于 emacs 的详细使用方法，读者可以查阅相关的资料，在此不再赘述。

除了以上介绍的 vi 和 emacs 编辑器外，在 Ubuntu X Window 系统中还提供了 gedit 编辑器，这个编辑器跟 Windows 下的 EditPlus 类似，如图 2-5 所示。读者可以根据自己的爱好选择相应的编辑器。



图 2-5　gedit 编辑器使用界面

2.4　编译器 gcc 的使用

gcc(GNU C Compiler)是 GNU 推出的功能强大、性能优越的多平台编译器，使用 gcc 可以编译 C 和 C++源代码，编译出的目标代码质量非常好，编译速度也很快。本节主要讨论 gcc 的安装与使用。

2.4.1　Ubuntu 下 gcc 的安装与设置

在安装 Ubuntu 7.10 时，gcc 是默认安装的。但在刚安装完系统时，系统中的 gcc 并不能用来开发，还缺少常用的头文件和库文件，还需要安装 build-essential 这个包。把 Ubuntu 系统的安装盘装入光驱或者把系统接入网络，在终端下执行下列命令：

```
lxy@lxy-desktop:/etc/apt$ sudo apt-get install build-essential
```

系统会提示输入密码，在输入密码后，系统会自动安装编译所需要的相关文件。系统在安装时，会把程序文件放入以下几个目录。

● /usr/lib

大部分的编译程序放在这个目录中。在这里有编译时需要的可执行程序，还有一些特定版本的库文件与头文件等。

● /usr/bin/gcc

指的是编译程序，即实际在命令行中执行的程序。这个目录可供各个版本的 gcc 使用，只要用不同的编译程序目录来安装就可以。

- /usr/include

这个目录及其子目录下包含程序所需要的头文件。缺少头文件，gcc 在编译时会出现找不到头文件的错误。

在安装完成之后，可以查看 gcc 的版本，在终端下输入 gcc –v，系统相应的响应如下：

```
lxy@lxy-desktop:~$ gcc -v
使用内建 specs。
目标：i486-linux-gnu
配置为：../src/configure -v --enable-languages=c,c++,fortran,objc,obj-c++,treelang --prefix=/usr
        --enable-shared --with-system-zlib --libexecdir=/usr/lib --without-included-gettext
        --enable-threads=posix --enable-nls --with-gxx-include-dir=/usr/include/c++/4.1.3
        --program-suffix=-4.1 --enable-__cxa_atexit --enable-clocale=gnu --enable-libstdcxx-debug
        --enable-mpfr --enable-checking=release i486-linux-gnu
线程模型：posix
gcc 版本 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
```

上面的信息说明 gcc 的版本是 4.1.3。目前 gcc 仍然处于不断完善与更新之中，每隔几个月就会有新的稳定发行版本产生。读者可以通过访问 http://www.gnu.org/software/gcc/来了解 gcc 的最新发展，下载最新的软件套件。

在确定系统中存在 gcc 及其版本后，打开编辑器输入下列代码：

```
#include   <stdio.h>
main()
{
        printf("Hello world!\n");
 }
```

编辑完成后，命名为 test.c，保存到当前目录下，然后输入下列命令：

```
lxy@lxy-desktop:~$ gcc -o test test.c
lxy@lxy-desktop:~$ ./test
Hello world!
```

其中第 1 个命令是对 test.c 文件进行编译链接，-o 选项的功能是指定创建可执行文件的名称，这里为 test，第 2 个命令是运行当前目录下的 test 程序。最后一行是程序输出。程序如果能成功执行，说明 gcc 及其相应的编译环境已经安装成功。

2.4.2 gcc 的使用

在上面用 gcc 编译 C 程序生成可执行文件的过程中，看起来像是一步就完成了，但实

际上它要经历如下的四个步骤：

(1) 预处理：这一步需要分析各种命令，如#define、#include、#if 等。gcc 调用 cpp 程序来进行预处理工作。

(2) 编译：这一阶段根据输入文件产生汇编语言，由于通常是立即调用汇编程序，所以其输出一般不保存在文件中。gcc 调用 ccl 进行编译工作。

(3) 汇编：这一步将汇编语言用作输入，产生具有.o 扩展名的目标文件。gcc 调用 as 进行汇编工作。

(4) 链接：这一阶段中，各目标文件被放在可执行文件的适当位置上，该程序引用的函数也放在可执行文件中(对使用共享库的程序稍有不同)。gcc 调用链接程序 ld 来完成最终的任务。

和大多数 shell 命令一样，gcc 的基本使用方式是：

```
gcc [选项] 文件名
```

gcc 可以通过选项对程序的生成进行全面的控制，每个选项可以有多种取值，在此只对其中常用部分进行介绍，其余的参数可以参考 gcc 手册或其他专门资料。gcc 的常用选项如表 2-14 所示。

表 2-14　gcc 常用选项说明

选　　项	说　　明
-c	仅对源文件进行编译，不链接生成可执行文件。在对源文件进行编译，或只需产生目标文件时可以使用该选项
-o filename	将经过 gcc 处理过的结果存为 filename，这个结果文件可以是预处理文件、汇编文件、目标文件或者最终的可执行文件。假设被处理的源文件为 file1，如果这个选项被忽略，那么生成的可执行文件默认名称为 a.out；目标文件默认名为 file1.o；汇编文件默认名为 file1.s；生成的预处理文件则发送到标准输出设备 stdout
-g 或-gdb	在可执行文件中加入调试信息，方便进行程序的调试。如果使用-gdb 选项，表示加入 gdb 扩展的调试信息，以便使用 gdb 来进行调试
-O[0、1、2、3]	对生成的代码进行优化，括号中的部分为优化级别，默认的情况为 2 级优化，0 为不优化。优化和调试通常不兼容，同时使用-g 和-O 选项经常会使程序产生奇怪的运行结果。所以不要同时使用-g 和-O 选项
-Dmacro[=def]	将名为 macro 的宏定义为 def，如果括号中的部分为默认值，则宏被定义为 1
-Umacro	某些宏是被编译程序自动定义的。这些宏通常可以指定在其中进行编译的计算机系统类型的符号，用户可以在编译某程序时加上-v 选项以查看 gcc 默认定义了哪些宏。如果用户想取消其中某个宏定义，用-Umacro 选项，这相当于把#undef macro 放在要编译的源文件的开头
-Idir	将 dir 目录加到搜寻头文件的目录列表中去，并优先于 gcc 默认的搜索目录。在有多个-I 选项的情况下，按命令行上-I 选项的前后顺序搜索。dir 可使用相对路径

(续表)

选　　项	说　　明
-Ldir	将 dir 目录加到搜寻-L 选项指定的函数库文件的目录列表中去，并优先于 gcc 默认的搜索目录。在有多个-L 选项的情况下，按命令行上-L 选项的前后顺序搜索。dir 可使用相对路径
-lname	在链接时使用函数库 name.a，链接程序在-Ldir 选项指定的目录下和/lib、/usr/lib 目录下寻找该库文件。在没有使用-static 选项时，如果发现共享函数库 name.so，则使用 name.so 进行动态链接
-static	禁止与共享函数库链接
-shared	尽量与共享函数库链接，这是链接程序的默认选项

gcc 的命令选项可以组合使用，不过在使用时，每个命令选项都要有一个自己的连字符“-”。如果采用简写的方式，很可能使命令的含义完全不同。

在 Linux 下生成的可执行文件没有固定的扩展名。任何符合 Linux 要求的文件名，只要文件的访问属性中有可以执行的属性，该文件就是可以执行的。因此，在使用上面介绍的-o filename 参数时，如果是生成链接后的可执行文件，filename 变量可以取任意一个符合 Linux 要求的文件名。

gcc 命令中的第 2 部分是一个输入给 gcc 命令的文件。gcc 按照命令选项的要求对输入文件进行处理，形成结果输出文件。输入的文件不一定是 C 的源代码文件，还可能是预处理文件、目标文件等。gcc 是通过输入文件的扩展名来确定输入文件的类型的。表 2-15 列出了 gcc 与 C/C++相关的输入文件扩展名命名规范。

表 2-15　gcc 文件扩展名规范

扩　展　名	类　　型
.c	C 语言源程序，可以被 gcc 预处理、编译、汇编、链接
.C, .cc, .cp, .cpp, .c++, .cxx	C++语言源程序，可以被 gcc 预处理、编译、汇编、链接
.i	预处理后的 C 语言源程序，可以被 gcc 编译、汇编、链接
.ii	预处理后的 C++语言源程序，可以被 gcc 编译、汇编、链接
.s	预处理后的汇编程序，可以被 as 汇编、链接
.S	未预处理的汇编程序，可以被 as 预处理、汇编、链接
.h	头文件，不进行任何操作
.o	编译后的目标文件，传送给 ld
.a	目标文件库，传送给 ld

在实际的开发过程中，很少有像上一节例子中给出的测试程序那样简单，为了使代码结构更合理，更方便进行代码的重用，通常采用将主函数和其他函数放在不同文件中的方

法。除了主程序之外，每个函数都由函数声明(函数头)和函数实现(函数体)两部分组成。函数的声明一般放在头文件(*.h)中，而函数的定义文件放在实现文件中(*.c)。gcc 可以很容易地把多个源文件编译成目标代码并进行链接。比如下列代码(test2.c)：

```
#include   <stdio.h>
#include "f2.h"
main()
{
        printf("Hello world!\n");
        f2();
}
```

在上段代码中，与 test.c 相比，增加了一条预处理语句(#include "f2.h")，并在主函数 mian 中增加了一个 f2 函数，它的代码如下：

```
/********    f2.c 文件内容   **************/
#include   <stdio.h>
void f2()
{
        printf("This is printed by f2!\n");
}
```

而 f2.h 提供了 f2 函数的原型，代码如下：

```
/********    f2.h 文件内容   **************/
void f2(void);
```

上述代码编辑完成之后，保存到当前目录下，现在要编译 test2.c 文件就要复杂一些，命令如下：

```
lxy@lxy-desktop:~$ gcc test2.c f2.c -o test2
lxy@lxy-desktop:~$ ./test2
Hello world!
This is printed by f2!
```

第 1 行命令是把 test2.c 和 f2.c 编译为 test2，第 2 行命令是执行 test2，第 3 行和第 4 行是执行结果。

通常一个项目会有很多源文件需要编译，显然像上面那样仅用一条 gcc 命令来完成编译工作是不现实的，而且当用户只是修改了其中某一个文件的时候，完全没有必要将每个文件都重新编译一遍。为了简化生成代码的步骤，熟练使用 gcc 还是不够的，GNU 提供了 make 这样的辅助工具。后面将介绍 make 的用法。

2.5 Linux C 程序的开发过程

对编辑器和 gcc 编译器有了基本的概念之后，接着便可以编写程序了。一个程序从最初的构思到最后完成的过程称为“开发过程”。C 语言程序的开发过程如下：

(1) 先根据问题要求，画出流程图，把程序设计好。

(2) 在文本编辑器中输入程序，并存成 C 源程序文件(扩展名一般为.c)。

(3) 执行编译器(compiler)，把源程序编译成目标程序。

(4) 执行链接器(linker)，链接由编译器产生的目标程序。编译器会链接所指定的各个目标文件及函数库文件，然后产生一个可执行文件。

(5) 执行产生的可执行文件。如果一切无误，程序便开发成功了。

(6) 如果发现错误，则再回到第(1)或第(2)步来修改程序，然后再次编译、链接……这个动作要一直重复，直到程序能正确执行为止。

(7) 后续的改进工作。

其中，步骤(1)与(7)是程序设计的工作，而步骤(2)至(6)则属于操作的部分。操作部分流程如图 2-6 所示。

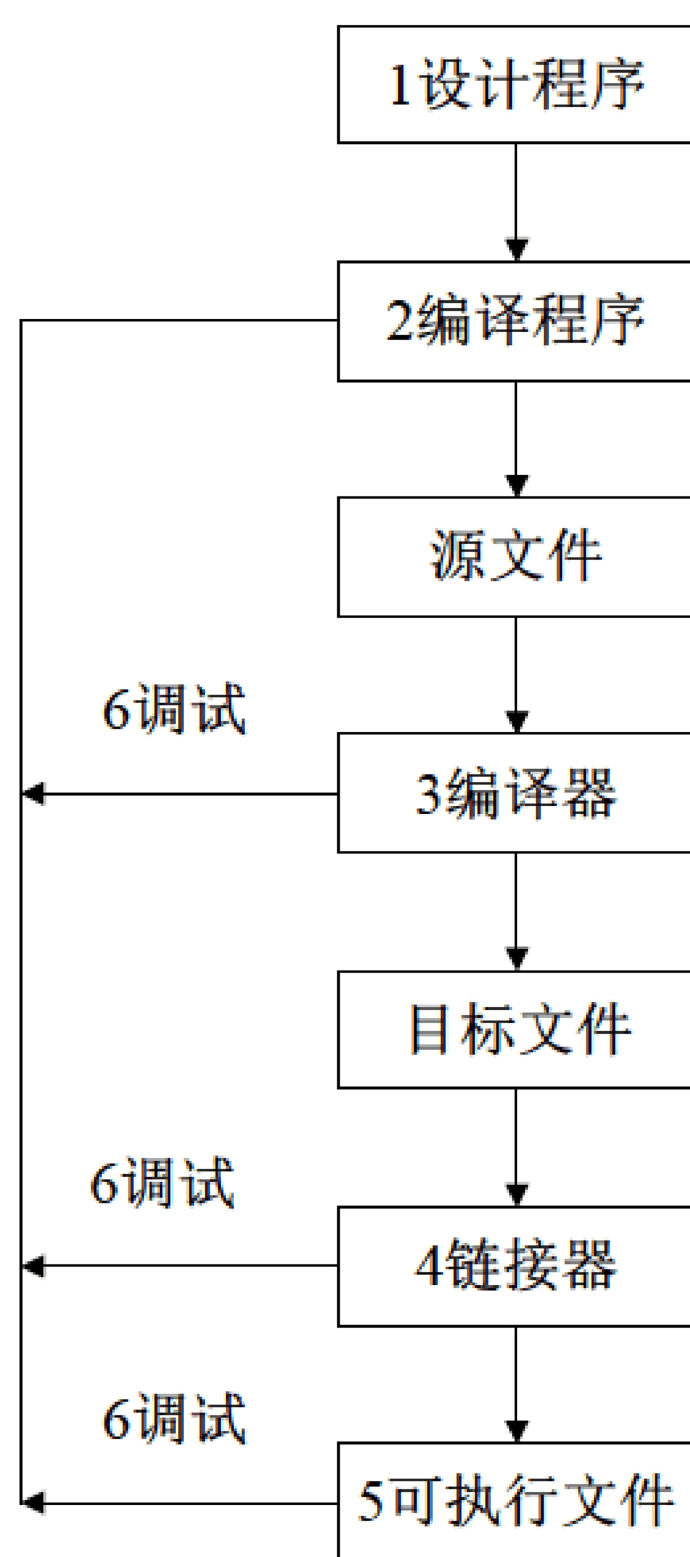


图 2-6　C 程序的开发过程

下面介绍如何从编辑一个 C 程序开始，完成编译、链接等操作。

2.5.1 编辑程序

首先，根据个人爱好，打开一个文本编辑器(vi、emacs、gedit 等)，以 gedit 为例，进入后，输入如下内容，如图 2-7 所示。



图 2-7 编辑 C 语言源程序

输入完成后，单击“保存”按钮，在弹出来的窗口中输入文件名 hello.c，然后单击“保存”按钮，如图 2-8 所示。



图 2-8 保存源文件

编辑好源程序后，必须使用编译器把源程序编译成计算机所熟悉的形式，这种计算机所熟悉的程序叫做目标程序。用户的程序编译成目标程序后，还要经过链接的环节与必要的模块链接后才能执行。这种经过编译、链接，可以执行的文件程序称为可执行文件。

在 Linux 中，可执行文件并没有特别的扩展名(如 Windows 规定一定要为.exe 或.com)，

一个文件能否执行，是以该文件是否具有被执行的权限来进行区分的，这个在第一章中已有过介绍。

2.5.2 编译程序

源程序编辑完成后，接下来要把源文件变成可执行文件。在 hello.c 文件所在的目录下，执行以下命令编译程序：

```
lxy@lxy-desktop:~/src/chapter2$ gcc hello.c -o hello
```

如果在编译、链接过程中发生错误，可能会出现如下的结果：

```
lxy@lxy-desktop:~/src/chapter2$ gcc hello.c
hello.c: 在函数 ‘main’ 中:
hello.c:6: 错误: expected ‘;’ before ‘}’ token
```

接着打开 hello.c 文件，原来第 5 行的尾部少了一个分号，如图 2-9 所示。



图 2-9 第 5 行尾部少了一个分号

加上分号后，保存退出。再次编译、链接：

```
lxy@lxy-desktop:~/src/chapter2$ gcc hello.c
```

假如一切无误，编译完成后，执行 ls 命令可以看到当前目录下多出一个 a.out 文件：

```
lxy@lxy-desktop:~/src/chapter2$ ls
a.out    hello.c
```

在执行 gcc 命令时，若不加任何参数，默认会产生名称为 a.out 的可执行文件(但不会产生.o 的目标文件)。此时，接着执行 a.out，看看执行的结果：

```
lxy@lxy-desktop:~/src/chapter2$ ./a.out
Hello, world!
```

关于 gcc 更多的参数说明，可以参考 2.4.2 节。

以上是一个简单的调试范例。其实，根据程序错误原因的不同，gcc 编译时会产生不同的警告或错误信息。而程序调试的技巧，是需要经过不断的练习，累积经验而来的。程序调试时，可以参考 2.9 节中的介绍，利用 gdb 来协助调试。

2.6 make 工具及其使用

当整个软件系统被划分为几个小的子系统，子系统又划分为几个独立工作的、由一组文件组成的模块时，就牵涉到模块之间的协调问题。在一个模块被修改以后，怎样才能保证其他模块与之相关的部分也随之改变，而不会影响模块之间的协调关系呢？make 就是用来进行协调的工具，它本身就是一个单独工作的程序，可以根据程序模块的修改情况重新编译链接目标代码，以保证目标代码总是由它的最新模块组成。本节将介绍 make 工具的使用。

2.6.1 make 命令和 Makefile

要使用 make，必须编写一个叫 Makefile 的文件。它描述了软件包中各个文件之间的关系，提供了更新每个文件的命令。在一个软件包里，通常是可执行文件由链接目标文件更新，而目标文件由编译源文件更新。

当一个适当的 Makefile 存在时，每次改变某些源文件，用简单的 shell 命令

```
make
```

将足以完成所有必需的重新编译。比如假设应用程序名为 exe1，它由 2 个目标代码模块组成，分别是 module1 和 module2。如果键入以下命令：

```
gcc module1.o module2.o –o exe1
```

生成了可执行文件 exe1，可用 make 来表示目标、依赖模块和命令的相关行为：

```
exe1:module1.o module2.o
        gcc module1.o module2.o –o exe1
```

以上命令说明 exe1 是目标文件，它依赖于 2 个模块：module1 和 module2。生成 exe1 后如果这些模块中任何一个改变了，就要执行相关行的编译命令。

若 module1.o 依赖于 module1.c 和头文件 module1.h，module2.o 依赖于 module2.c 和 module2.h，这些依赖关系可以写成：

```
module1.o:modul1.c module1.h
        gcc –c module1.c
```

```
module2.o:module2.c module2.h
        gcc –c module2.c
```

make 程序利用 Makefile 的数据和每个文件最新一次更改的时间来确定哪些文件需要更新；对每个需要更新的文件，make 程序使用 Makefile 中定义的命令来更新它。Makefile 文件需要按照某种语法进行编写，在文件中说明如何编译各个源文件并链接生成可执行文件，并要求定义源文件之间的依赖关系。Makefile 的每个相关行说明一个目标依赖于哪几个文件，以及生成或更新目标时所需要的命令。Makefile 文件的格式如下：

```
目标：依赖项列表
        [命令]
```

其中，“依赖项”一般为生成目标所需的其他目标或者文件名。如，在上例中，生成最终可执行文件需要 module.o 和 module2.o。“命令”为生成目标所需执行的 gcc 命令，其中，“命令”所在行的行首要有空格，空的格数为一个制表位(Tab)。Makefile 文件也可以在描述语句行前面加“#”表示注释，make 程序将跳过此行不执行。相关行如果过长，还可以使用反斜线“\”作为后接换行符来续行。

make 程序执行 Makefile 的相关行的默认情况是将执行状态显示出来，如果在相关行前加“@”，就可以避免显示该行。

为了保证 make 程序正常工作，需要把 Makefile 文件放在与源程序相同的目录下。如果 make 程序没有使用-f 选项指定一个 Makefile，make 将在当前目录下按顺序寻找下列文件：GNUMakefile、Makefile 和 makefile。推荐使用 Makefile，因为它的第一个字母是大写，通常被列在一个目录文件列表的最前面。Makefile 文件都可以用 vi 或 emacs 这样的文本编辑器来编辑。以前面给出的 exe1 程序为例，下面列出了一个完整的 Makefile 文件。

```
exe1 : module1.o module2.o
        gcc module1.o module2.o –o exe1
module1.o : modul1.c module1.h
        gcc –c module1.c
module2.o : module2.c module2.h
        gcc –c module2.c
clean :
        rm –f exe1 *.o
```

在 Makefile 文件中除了依赖关系的描述外，还可以含有宏。宏是代表文件名和命令任选项的短名，后面会讨论。

建立了 Makefile 文件，就可以使用 make 程序创建或更新 Makefile 文件中的目标，其命令格式如下：

```
make [选项] [宏] [目标]
```

其中，选项是定义 make 如何工作的命令选项；宏是执行 make 时使用的宏值；目标是

需要更新的目标名，这个目标必须在 Makefile 文件中已经给出描述。这些参数可以根据情况选定或不定义而使用默认值。make 命令的常用选项如表 2-16 所示。

表 2-16　make 常用选项说明

扩　展　名	类　　型
-f	指定 Makefile 文件名
-p	打印出 Makefile 中所有宏定义和描述内部规则的相关行
-i	忽略 Linux 命令返回的错误，继续执行下面的命令。如果没有该选项，则遇到 Linux 命令出错就会停止
-s	表示执行而不显示执行状况
-r	忽略内部规则
-n	按实际运行时的执行顺序显示命令，包括以“@”打头的命令，但并不真正执行。这个选项常用来检查 Makefile 文件的正确性
-d	Debug 模式，输出有关文件和检测时间的详细信息
-t	修改每个目标文件的更新日期，但不重写创建这些文件
-c dir	在读取 Makefile 之前改变到指定的目录 dir
-I dir	指定使用的 Makefile 所在的目录
-w	在处理 Makefile 之前和之后，都显示工作目录

在命令行中，如果只输入

```
make
```

而未指定其他任何参数，make 将对 Makefile 中的第一行目标进行维护。在前面的例子中，按照以上默认规则，就应该将 exe1 作为目标来进行维护。在发现目标依赖于其他文件时，又继续在 Makefile 文件中寻找以新的依赖文件为目标的相关的文件，并这样层层进行搜索。

make 程序也可以指定要进行维护的目标，比如：

```
make module1.o
```

就只把 module1.o 当作目标，而只考虑它所依赖的文件的更新。如果想使用自己指定的 Makefile 文件，可以使用如下命令：

```
make –f filename
```

这样，make 就在当前目录下寻找文件名为 filename 的 Makefile 文件，并读入该文件的相关行。

2.6.2　Makefile 的规则

有时，由于在开发过程中没有详细地划分源文件的模块，所以并不非常确定源文件的

相互依赖关系。因此在编写完源程序文件后，需要从中生成需要的 Makefile 规则。最基本的编写规则的方法是从最终的源程序文件开始一个一个地查看源码文件。把它们要生成的目标文件作为目标，而 C 语言源码文件和源码文件包含的头文件作为依赖文件生成规则。但是我们必须去分析源码文件的套嵌关系，比如需要把某些头文件包含的头文件也作为依赖文件，当文件很多时，这种方法是很繁琐的，也不能保证其正确性。

用户需要的是自动从源码文件中产生文件的相互依赖关系，编译器可以做这个工作。当编译器编译每一个源码文件的时候，它知道应该包括什么样的头文件。当使用 gcc 的时候，用-M 开关，它可以为每一个输入的 C 语言源文件输出一个依赖规则，把 gcc 将要生成的目标文件作为 Makefile 规则的目标文件，而把生成这个目标文件的 C 语言源文件和所有应该被引用的头文件作为依赖文件。

需要注意的是，这种方法中，gcc 并不区分系统的头文件和程序自带的头文件，规则的依赖文件列表中将包括被角括号("<"，">")和双引号(" " ")所标注的头文件。由于在一般情况下不会修改系统头文件，为了避免输出的依赖关系中包含系统头文件，可以用-MM 参数来代替-M 传递给 gcc。

gcc 只输出规则的依赖关系，不含有命令部分。用户可以自己写入需要的命令，或者什么也不写，make 会使用隐含规则。

2.6.3 Makefile 中的变量

Makefile 里的变量就像一个环境变量。事实上，环境变量在 make 中也被解释成 make 的变量。这些变量对大小写敏感，一般使用大写字母。几乎可以从任何地方引用定义的变量，变量的主要作用如下：

- 保存文件名列表。在前面的例子里，作为依赖文件的一些目标文件名出现在可执行文件的规则中，而在这个规则的命令行里同样包含这些文件并传递给 gcc 作为命令参数。如果使用一个变量来保存所有的目标文件名，则可以方便地加入新的目标文件而且不易出错。
- 保存编译器的参数。在很多源代码编译时，gcc 需要很长的参数选项，在很多情况下，所有的编译命令使用一组相同的选项，如果把这组选项使用一个变量代表，那么可以把这个变量放在所有引用编译器的地方。当要改变选项的时候，只需改变一次这个变量的内容即可。

Makefile 中的变量是用一个字符串在 Makefile 中定义的，这个文本串就是变量的值。只要在一行的开始写下这个变量的名字，后面跟一个＝号，然后跟要设定的这个变量的值即可定义变量，下面是定义变量的语法：

```
变量名=字符串
```

使用时，把变量用括号括起来，并在前面加上$符号，就可以引用变量的值：

$(变量名)

make 解释规则时“变量名”在等式右端展开为定义它的字符串。变量一般都在 Makefile 的头部定义。按照惯例，所有的 Makefile 变量都应该大写。如果变量的值发生变化，就只需要在一个地方修改，从而简化了 Makefile 的维护。

现在利用变量把前面的 Makefile 重写一遍：

```
OBJS=module1.o module2.o
C= -c
exe1 : $(OBJS)
        gcc $(OBJS) –o exe1
module1.o : modul1.c module1.h
        gcc C module1.c
module2.o : module2.c module2.h
        gcc C module2.c
clean :
        rm –f exe1 *.o
```

Makefile 中定义了一些默认变量，这些变量具有特殊的含义，可在规则中使用。表 2-17 给出了 Makefile 中一些主要的默认变量。

表 2-17　Makefile 中的默认变量

选　　项	说　　明
AR	归档维护程序的名称，默认值为 ar
ARFLAGS	归档维护程序的选项
AS	汇编程序的名称，默认值为 as
ASFLAGS	汇编程序的选项
CC	C 语言编译器的名称，默认值为 cc
CFLAGS	C 语言编译器的选项
CPP	带有标准输入的 C 语言预处理程序，默认值为 $(CC) -E
CPPFLAGS	C 语言预编译的选项
RM	删除文件的命令，默认值为 rm –f
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$<	第一个依赖文件的名称
$?	所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚
$@	目标的完整名称
$^	所有的依赖文件，以空格分开，不包含重复的依赖文件
$%	如果目标是归档成员，则该变量表示目标的归档成员名称

2.6.4 伪目标

在 Makefile 中，并不是所有的目标都对应于磁盘上的文件，有的目标的存在只是为了形成一条规则，从而使 make 完成特定的工作，并不生成新的目标文件，这样的目标称为伪目标。常用的伪目标有 all、clean 等。例如：

```
all : exe1 exe2 exe3
exe1 : exe1.c exe1.h
        gcc exe1.c –o exe1
exe2 : exe2.c exe2.h
        gcc exe2.c –o exe2
exe3 : exe3.c exe3.h
        gcc exe3.c -o exe3
clean :
        rm –f exe*
```

其中，all、clean 即为伪目标，一个伪目标和一个正常的目标几乎是一样的，只是这个目标文件不存在。以上 Makefile 中的第一条规则下的命令行为空，make 不会执行任何动作，只是检查依赖文件的更新情况，所以会扫描剩下的几条规则并执行相应的编译命令生成可执行文件。

同样，由于没有任何其他的规则依赖 clean，因此在命令行执行 make 时，这条规则将不会被执行。但是，如果明确地使用命令 make clean，make 会把命令行上的参数 clean 作为它的目标，并执行对应的删除命令。

2.6.5 条件语句

条件语句可以将一个变量与其他变量的值进行比较，或将一个变量与一字符串常量相比较。这样就可以根据变量的值执行或忽略 Makefile 文件中的一部分脚本。条件语句用于控制 make 实际看见的 Makefile 文件部分，不能用于在执行时控制 shell 命令。

条件语句包含 3 条指令：ifeq、else 和 endif。

ifeq 指令是条件语句的开始，它还指明了条件。它包含 2 个参数，参数之间被逗号分开，并被括在圆括号内。运行时首先对 2 个参数变量替换，然后进行比较。在 Makefile 中跟在 ifeq 后面的行是符合条件时执行的命令；不符合条件时，它们将被忽略。

else 指令则用来说明如果前面的条件不满足，将执行跟在其后面的命令执行。在条件语句中，else 指令是可选择使用的。

endif 指令结束条件语句。任何条件语句必须以 endif 指令结束，后跟 Makefile 文件中的正常内容。

例如下列条件语句：

```
ifeq( $(VAR),1)
        gcc –o exe1 module
else
        gcc –o exe2 module
endif
```

上述条件语句说明在变量 VAR=1 时，把 module 模块编译输出文件为 exe1，不等于 1 时，把 module 模块编译输出文件为 exe2。

2.6.6　调试 make

由于 make 的规则并不像源程序代码那样直观，而且 Makefile 也无法像 C 语言程序那样进行联机调试。因此初学者经常对 make 的错误感到困惑。实际上，如果在使用 make 遇到问题时，可以通过-d 选项使 make 在执行命令时打印调试信息，这些信息包括以下内容：

- make 重新编译时需要检查的文件。
- 哪些文件被比较以及比较的结果。
- 需要重新生成的文件。
- make 将要使用的隐含规则。
- make 实际执行的隐含规则以及命令。

读者需要通过不断地练习并参考其他 Makefile 的例子才能熟练地掌握 make 工具的使用。

2.7　使 用　autoconf

autoconf 是一个用于生成可以自动配置软件源代码包以适应多种类 Unix 系统的 shell 脚本的工具。由 autoconf 生成的配置脚本在运行的时候与 autoconf 是无关的，就是说配置脚本的用户并不需要拥有 autoconf。

由 autoconf 生成的这些脚本通常被命名为 configure，它们检查当前系统是否满足软件正常运行所需要的特征，并根据检查得到的信息生成 Makefile。使用者所需要做的只是在软件发布版本的源程序目录中执行./configure，剩下的工作不需要用户的手工干预，配置脚本可以自动地确定系统的类型。还可以对软件包可能需要的各种特征进行独立的测试。

对于使用了 autoconf 的软件包，autoconf 根据软件包需要的系统特征，来生成相应的配置脚本，如果系统特征变动，只要再次运行 autoconf 就可以再次生成相应的配置脚本。

为了生成配置脚本，autoconf 需要宏处理工具 GNU m4。autoconf 使用了某些 Unix 版本的 m4 所不支持的特征，它还使用了包括 GNU m4 1.0 在内的某些以往版本没有的扩展功能。因此，为了运行 autoconf，必须使用 GNU m4 的 1.1 版或者更新的版本。可以发现，使用比 1.1 或 1.2 更新的版本将比使用 1.1 或 1.2 版快许多。Ubuntu 7.10 系统带的 m4 的版

本是 1.4。

2.7.1 创建 configure 脚本

由 autoconf 生成的配置脚本通常被称为 configure。运行的时候，configure 会创建一些文件。在这些文件中，配置参数被适当的值所替换。由 configure 创建的文件有：

- 一个或者多个 Makefile 文件，在包的每个子目录中都有一个。
- 有时创建一个 C 头文件，它的名字可以被配置，该头文件包含一些#define 语句。
- 名为 config.status 的 shell 脚本，在运行时，它将重新创建上述文件。
- 名为 config.cache 的 shell 脚本，它保存了许多测试的运行结果。
- 名为 config.log 的文件，包含了由编译程序输出的许多信息，以便在 configure 出现错误时进行调试。

为了使用 autoconf 创建 configure 脚本，首先需要编写一个 configure.in 文件，并作为 autoconf 的输入文件运行 autoconf。如果 autoconf 所提供的预定义测试不能满足要求，则需要自行编写特征测试，这时可能还要编写一个名为 aclocal.m4 的文件和一个名为 acsite.m4 的文件。如果在软件包中使用了包含#define 语句的 C 语言头文件，可能还需要编写 acconfig.h，此时 autoconf 会生成一个文件 config.h.in，在软件包发布时，需要包含这个文件。

2.7.2 编写 configure.in 文件

为了为软件包创建 configure 脚本，需要编写一个名为 configure.in 的文件，该文件包含了对软件包需要或者可以使用的系统特征进行测试的 autoconf 宏的调用。现有的 autoconf 宏可以检测类 Unix 系统的许多特征。对于大部分其他特征，可以使用 autoconf 提供的模版宏来创建自己定制的测试。为了检测某些非常特殊的特征，有时不得不在 configure.in 文件中手工编写一些 shell 命令。使用程序 autoscan 可以简化编写 configure.in 的工作。

除了少数特殊情况之外，在 configure.in 中可以用任意的次序调用 autoconf 定义的测试宏。但是在每个 configure.in 中，必须在进行任何测试之前调用 AC_INIT 宏，并且在结尾处包含一个对 AC_OUTPUT 的调用，只有这两个宏是必需的，而其他宏可以根据需要添加。

此外，有些宏要求其他的宏在它们之前被调用，这是因为这些宏需要用到前面测试所得到的值以决定执行的动作。如果在 configure.in 中没有按照规定的顺序调用宏，在生成配置脚本 configure 时会发出警告信息。

通常，在列表后面的测试往往依赖于前面的测试。例如，库函数可能受到 typedefs 和库的影响。函数库的存在与否直接影响到是否可以包含相应的头文件，所以对头文件的检查要放在检查完函数库之后。还有，一些系统服务需要用到某些特殊库函数，而只有当在某些头文件中声明了函数原型定义后才可以调用这些库函数，另一方面，如果所需的函数

库不存在，程序就不能在头文件中调用这些原型函数。

configure.in 中的宏调用次序一般如下所示：

```
AC_INIT(file)
程序选择测试
检测库文件
检测头文件 (.h)
检测 typedefs
检测 structures
检测编译器特征
检测库函数
检测系统服务
AC _OUTPUT([file…])
```

一般来说，除非能肯定改动这些测试的意义，并且有足够的理由进行这样的改动，否则最好不要改变上表中所建议的宏的调用次序。下面为一个 configure .in 文件的实例。

```
AC_INIT(configure.in)
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(testc, 0.1)
AC_LANG_C
AC_PROG_CC
AM_PROG_LIBTOOL
AC_OUTPUT(Makefile src/Makefile)
```

每个 autoconf 生成的 configure 脚本必须以调用宏 AC_INIT 为开始，以对 AC_OUTPUT 的调用结尾。一般情况下，每个宏调用在 configure.in 中占据单独的一行。大部分宏在一行中即可完成需要的工作，并且以在宏调用之后的新行为结束符。这样，生成的 configure 脚本就不会存在大量的空行，使得脚本比较容易阅读。在宏调用的同一行中设置 shell 变量通常是允许的，这是因为 shell 允许出现不用新行间隔的赋值语句。

在调用带参数的宏的时候，在宏名和左括号之间不能出现任何空格。如果参数被 m4 引用字符“[”和“]”所包含，参数就可以多于一行。如果有一个长行，比如一个文件名列表，通常可以在行的结尾使用反斜线以便在逻辑上把它与下一行进行连接。

有些宏处理要对检测的结果进行判断，这分为两种情况：如果满足了某个给定的条件就执行特定的动作，如果没有满足某个给定的条件就执行另外的动作。在某些情况下，可能希望在条件为真的情况下执行某些动作，在为假时什么也不作，反之亦然。为了忽略为真的情况，把空值作为参数 action-if-found 传递给宏，为了忽略为假的情况，可以忽略包括前面的逗号在内的宏的参数 action-if-not-found。

可以在 configure.in 中添加注释。注释以 m4 预定义宏 dnl 开始，该宏丢弃在下一个新行之前的所有脚本。这些注释并不在生成的 configure 脚本中出现。例如，可以在文件 configure.in 的开头处包含以下的信息：

```
dnl Process this file with autoconf to produce a configure script
dnl
dnl
dnl Author : lxy
```

2.7.3 使用 autoscan 创建 configure.in 文件

在大多数情况下，不需要在 configure.in 文件中手工输入复杂的宏定义，用脚本程序 autoscan 可以简化为软件包创建 configure.in 文件的工作。autoscan 是 autoconf 软件包提供的 Perl 脚本程序，它从源程序中抽取与函数调用和头文件有关的信息，并将其输出到 configure.scan 文件中。如果在命令行中给出了目录参数，autoscan 就在给定目录及其子目录中检查源文件；如果没有给出目录，就在当前目录及其子目录中进行检查。它对源文件进行检查，寻找软件代码的移植性问题并创建文件 configure.scan，该文件可以作为软件包的 configure.in 文件的初始版本。

在把 configure.scan 文件直接改名为 configure.in 之前应该进行手工检查。可能需要做一些调整才能形成最后版本的 configure.in 文件。在某些并不多见的情况下，autoscan 在输出的宏列表中会把某些宏的顺序搞错，autoconf 在运行时会给出警告信息，因此需要手工地安排某些宏的顺序。

另外，如果希望软件包使用一个配置头文件，则必须调用名为 AC_CONFIG_HEADER 的宏。可能还需要在程序中修改或者添加一些#if 指令使得程序可以与 autoconf 配合使用。

autoscan 要用到一些数据文件，这些数据文件是随 autoconf 的宏文件一起安装的，以便在包中的源文件中发现某些特定符号时决定输出这些宏。这些文件都具有相同的格式，每一个都是由符号、空格和在符号出现时应该输出的 autoconf 宏所组成的。以“#”号开头的行是注释。

2.7.4 用 autoconf 创建 configure

有了 configure.in 文件后，就可以开始生成自动配置脚本 configure 了。要生成 configure，只需要不带参数地运行程序 autoconf，autoconf 将使用 m4 宏处理器处理 configure.in 文件。如果提供了文件参数，autoconf 将读入指定的文件而不是默认的 configure.in 文件，并且把配置脚本输出到文件或设备而不是 configure 中。如果为 autoconf 提供参数“-”，它将从标准输入，而不是 configure.in 中读取并且把配置脚本输出到文件或设备。

autoconf 的宏定义在几个文件中，有些文件是与 autoconf 一同发布的。autoconf 运行时首先读入这些文件，然后在包含了发布的 autoconf 宏文件的目录中寻找可能出现的文件 acsite.m4，并且在当前目录中寻找可能出现的文件 aclocal.m4。除了 autoconf 中所定义的宏，还可以把其他需要的宏定义放在这些文件中，如果宏在多个文件中被重复定义，那么后面的定义将覆盖前面的定义。

2.7.5　更新 configure 脚本

如果有大量由 autoconf 生成的 configure 脚本，程序 autoconf 可以简化一些工作。它重复地运行 autoconf(在适当的情况下还运行 autoheader)以便重新创建以当前目录为根的目录树的 autoconf configure 脚本和配置头文件。在默认情况下，它只重新创建那些比对应的 configure.in 或 aclocal.m4 旧的文件。在文件没有被改变的情况下，autoheader 并不改变它的输出文件时间戳(timestamp)，这可以使工作量最小化。如果安装了新版本的 autoconf，可以使用选项-forece 调用 autoconf 而重新创建所有的文件。

在同一个目录树中，autoconf 不支持两个目录作为同一个大包的一部分，也不支持每个目录都是独立包。如果使用了-localdir 选项，它假定所有的目录都是同一个包的一部分；如果没有使用-localdir，它假定每个目录都是一个独立的包。

2.8　使 用 automake

Makefile 基本构造虽然简单，但是刚开始学习写 Makefile 时会感到没有规范可循，每个人写出来的 Makefile 都不太一样，不知道从何下手，而且常常会受限于自己的开发环境，只要环境变量不同或路径改一下，可能就得修改 Makefile。虽然目前根据“GNU Makefile 惯例”制订出一些使用 GNU 程序设计时写 makefile 的标准和规范，但是内容很长而且很复杂，并且需要经常做些调整，为了减轻编写 Makefile 的工作，有了 Automake。

Automake 是一个从文件 Makefile.am 中自动生成 Makefile.in 文件的工具。每个 Makefile.am 文件是一系列 make 的宏定义，有时也会包含 make 规则。利用 automake 生成的 Makefile.in 服从 GNU Makefile 标准。典型的 automake 输入文件是一系列简单的宏定义。automake 将处理所有这样的文件以创建 Makefile.in。在一个项目的每个目录中通常包含一个 Makefile.am。

automake 在一些方面对项目做了假定，例如它假定项目使用自动配置工具 autoconf，并且对 configure.in 的内容也有某些限制。

为生成 Makefile.in，automake 需要 perl。但是由 automake 创建的软件发布完全服从 GNU 标准，并且在创建中不需要 perl。

2.8.1　automake 的工作流程

automake 首先读入 Makefile.am 文件，然后生成 Makefile.in。automake 根据 Makefile.am 中定义的宏和目标产生更多特定代码，例如，一个 bin_PROGRAMS 宏定义将生成一个需要被编译、连接的目标。automake 把 Makefile.am 中的宏定义和目标复制到生成的文件中。

开发者可以根据需要把任何代码添加到生成的 Makefile.in 文件中。需要注意的是 automake 不能识别 GNU 对 make 的扩展。在 Makefile.am 中使用这些扩展特性将导致错误或不确定行为。

Makefile.am 中定义的目标将会覆盖所有由 automake 自动生成的拥有相同名字的目标。虽然 automake 支持这一功能，但最好避免使用它，因为有些时候生成的规则非常难于理解。

类似地，在 Makefile.am 中定义的变量将覆盖任何通常由 automake 创建的变量定义，这个功能经常使用。但需要注意的是，许多由 automake 生成的变量都是内部使用的，并且它们的名字可能在以后的版本中改变。

2.8.2 使用 automake 生成 Makefile.in

本节仍以前面的 test 程序为例，介绍通过 automake 生成 Makefile.in 文件的过程。

首先，在当前工作目录下创建一个 test 目录，用它来存放 test 程序及相关文件。

```
lxy@lxy-desktop:~$ mkdir   test
lxy@lxy-desktop:~$cd   test
```

用自己喜欢的编辑器编辑如下文件：

```
#include   <stdio.h>
main()
{
        printf("Hello world!\n");
 }
```

编辑完成后，命名为 test.c 并保存到 test 目录下。接下来就要使用 automake 来产生 Makefile 文件。先使用 autoscan 命令根据目录下的源代码生成一个 configure.in 的模板文件。命令如下：

```
lxy@lxy-desktop:~/test$ ls
test.c
lxy@lxy-desktop:~/test$ autoscan
lxy@lxy-desktop:~/test$ ls
autoscan.log   configure.scan   test.c
```

执行之后，test 目录下就会产生一个文件：configure.scan，可以用它作为 configure.in 的蓝本。现在将 configure.scan 改名为 configure.in，并且编辑它，按下面的内容修改，去掉无关的语句：

```
# -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.61)
```

```
AC_INIT(test.c)
AM_INIT_AUTOMAKE(test,1.0)
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_OUTPUT(Makefile)
```

然后执行 aclocal 和 autoconf，分别会产生 aclocal .m4 以及 configure 文件：

```
lxy@lxy-desktop:~/test$ aclocal
lxy@lxy-desktop:~/test$ autoconf
lxy@lxy-desktop:~/test$ ls
aclocal.m4   autom4te.cache   autoscan.log   configure   configure.in   test.c
```

下面要做的工作是新建一个 Makefile.am 文件，同样根据自己的喜好选择相应的编辑器，输入下列内容：

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS=test
test_SOURCES=test.c
```

automake 会根据所写的 Makefile.am 来自动生成 Makefile.in。Makefile.am 中定义的宏和目标会指导 automake 生成指定的代码。例如，宏 bin_PROGRAMS 将导致编译和链接的目标被生成。

下一步将运行 automake，命令如下：

```
lxy@lxy-desktop:~/test$ automake --add-missing
configure.in:6: installing `./missing'
configure.in:6: installing `./install-sh'
Makefile.am: installing `./depcomp'
```

automake 会根据 Makefile.am 文件产生一些文件，包含最重要的 Makefile.in。执行 configure 生成 Makefile，命令如下：

```
lxy@lxy-desktop:~/test$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
```

```
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
lxy@lxy-desktop:~/test$ ls -l Makefile
-rw-r--r-- 1 lxy lxy 16574 2008-01-10 20:27 Makefile
```

此时，Makefile 已经产生出来了。现在就可以使用 Makefile 编译代码了：

```
lxy@lxy-desktop:~/test$ make
gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\"
-DPACKAGE_STRING=\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"test\"
-DVERSION=\"1.0\" -I.     -g -O2 -MT test.o -MD -MP -MF .deps/test.Tpo -c -o test.o test.c
mv -f .deps/test.Tpo .deps/test.Po
gcc   -g -O2    -o test test.o
```

运行 test：

```
lxy@lxy-desktop:~/test$ ./test
Hello world!
```

这样，test 就编译出来了。

以上简要介绍了 automake 的使用方法，关于更详细的使用方法，读者可以查阅 automake 的帮助手册。

2.9 使用 gdb 调试程序

即使是最优秀的程序员也不可避免地会在编程时出现一些这样或那样的错误。所有的程序在写好以后，都要经过调试，在调试过程中发现并改正程序中的错误。通常来说，软件项目的规模越大，调试起来就越困难，越需要一个强大而高效的调试器作为后盾。对于 Linux 程序员来说，目前可供使用的调试器非常多，gdb 就是其中较为优秀的一个，本节主

要介绍使用 gdb 调试程序的方法。

2.9.1 初次使用 gdb

排除编译、连接过程中的错误，只是程序设计中最简单、最基本的一个步骤。这个过程中的错误，只是我们在使用 C 语言描述一个算法中所产生的错误，是比较容易排除的。通常很多问题是在程序运行过程中所出现的，往往是算法设计有问题，需要更加深入地测试、调试和修改。一个稍微复杂的程序，往往要经过多次编译、连接及测试、修改。

Linux 包含了一个 gdb 的调试程序，gdb 是一个用来调试 C 和 C++程序的强力调试器，它使用户能在程序运行时观察程序的内部结构和内存的使用情况。gdb 提供了以下一些功能：

(1) 监视程序中变量的值。

(2) 设置断点以使程序在指定的代码行上停止执行。

(3) 一行行地执行代码。

在命令行上键入 gdb 并按 Enter 键就可以运行 gdb 了，如果一切正常，gdb 将被启动并且在屏幕上会看到如下类似内容：

```
lxy@lxy-desktop:~$ gdb
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(gdb)
```

启动 gdb 后，可以在命令行上指定很多的选项，也可以用下面的方式在命令行中指定想要调试的文件名。

```
lxy@lxy-desktop:~$ gdb filename
```

此时，gdb 会装入名为 filename 的可执行文件。下面以一个实例介绍如何一步步地用 gdb 调试程序。

程序 gdbtest 的作用是显示一个简单的“Hello World！”，再用反序将此输出。源代码如下：

```
#include <stdio.h>
void print1(char * string)
{
 printf("The string is %s\n",string);
```

```
}
void print2(char * string)
{
 char * string2;
 int     size, i;
 size=strlen(string);
 string2=(char *) malloc(size+1);
 for (i=0; i<size; i++)
         string2[size-i] = string[ i ];
   string2[size+1] ='\ 0' ;
   printf("The string printed backward is: %s\n", string2);
}

main()
{
 char test_string [ ] =" Hello World! ";
 print1(test_string);
 print2(test_string);
}
```

把上述代码保存为 gdbtest.c，用下面的命令编译它：

```
gcc -o gdbtest gdbtest.c
```

编译成功后，输入下列命令执行 gdbtest：

```
./gdbtest
```

这时，程序显示结果如下：

```
The string is    Hello World!
The string printed backward is:
```

从显示结果可以看出，程序输出的第一行是正确的，但第二行却没有按照我们的要求对“Hello World！”进行反序输出。这说明函数 print2 没有正常工作。下面要做的工作是通过 gdb 查找问题所在，为了使用 gdb 进行调试，命令行中必须加上-g 选项，这样可以使程序在编译时包含调试信息。调试信息包含程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。输入如下命令：

```
gcc gdbtest.c -g -o gdbtest
```

现在，可以使用 gdb 对 gdbtest 进行调试，输入如下命令：

```
gdb gdbtest
```

这个命令将载入 gdbtest 可执行文件，进入 gdb 后，输入 run 命令运行 gdbtest，结果显

示如下：

```
(gdb) run
Starting program: /home/lxy/gdbtest/gdbtest
The string is    Hello World!
The string printed backward is

Program exited with code 040.
```

程序的运行结果和在 gdb 外面运行的结果没有任何不同。为了找出问题所在，可以在 print2 函数的 for 语句后设一个断点。首先需要查找 for 语句所在的行号。在 gdb 下输入 list 命令可列出源代码：

```
(gdb) list
1        #include <stdio.h>
2        void print1(char * string)
3        {
4                printf("The string is %s\n",string);
5        }
6        void print2(char * string)
7        {
8                char * string2;
9                int        size, i;
10               size=strlen(string);
```

gdb 默认显示 10 行代码，在 gdb 提示符下按 Enter 键将重复上一个命令。此时按下 Enter 键，gdb 将再执行一次 list 命令，给出下列输出：

```
11               string2=(char *)malloc(size+1);
12               for (i=0; i<size; i++)
13                       string2[size-i]=string[i];
14       string2 [ size+1 ] ='\ 0' ;
15       printf( "The string printed backward is %s\n", string2);
16       }
17
18       main()
19       {
20               char test_string [ ] =" Hello World! ";
```

再按一次 Enter 键，将列出程序的剩余部分：

```
21               print1(test_string);
22               print2(test_string);
23       }
24
```

根据列出的源程序，能看到要设断点的地方在第 12 行。在 gdb 命令行提示符下输入如下命令设置断点：

```
(gdb) break 13
Breakpoint 1 at 0x8048457: file gdbtest.c, line 13.
```

现在再输入 run 命令，将产生如下的输出：

```
(gdb) run
Starting program: /home/lxy/gdbtest/gdbtest
The string is   Hello World!

Breakpoint 1, print2 (string=0xbfa4a901 " Hello World! ") at gdbtest.c:13
13                              string2[size-i]=string[i];
```

程序运行停止在第 13 行“string2[size-i]=string[i];”这条语句上。可以通过设置一个观察点来观察 string2[size-i]变量的值，看错误是怎样产生的。设置观察点输入如下命令：

```
(gdb) watch string2[size-i]
Hardware watchpoint 2: string2[size - i]
```

输入命令 c 使程序继续运行，直到停止在下次循环体语句处。经过第一次循环后，gdb 的显示如下：

```
(gdb) c
Continuing.
Hardware watchpoint 2: string2[size - i]

Old value = 0 '\0'
New value = 72 'H'
print2 (string=0xbfadd991 " Hello World! ") at gdbtest.c:12
12                      for (i=0; i<size; i++)
```

这个值正是所期望的。接着输入 c 命令执行循环，后来的数次循环的结果都是正确的。当 i=11 时，表达式 string2[size-i]的值等于'!'，size-i 的值等于 1，最后一个字符已经复制到新字符串里了。

如果继续执行循环，可以看到已经没有值分配给 string2[0]了，而它是新串的第一个字符，因为 malloc 函数在分配内存时把它们初始化为空(null)字符，所以 string2 的第一个字符是空字符。这就解释了为什么在输出 string2 时没有任何输出了。

问题出在“string2[size-i]=string[i];”这条语句上。C 语言字符串的起始偏移为 0，即字符串 string 的长度为 size 时，它的第一个字符是 string[0]，而不是 string[1]；它的最后一个字符应该是 string[size-1]，而不是 string[size]。本来期望将 string 的第一个字符放在 string2 的最后一个字符位置。但对“string2[size-i]=string[i]”语句来说，当 i=0 时，string[i]即为

string[0]，也就是 string 的第一个字符；但 string2[size-i]即为 string2[size]，并不是 string2 的最后一个字符，string2 的最后一个字符位置应该是 string2[size-1]。因此，“string2[size-i]=string[i]”语句应该修正为“string2[size-1-i]=string[i]”。另外，string2 的长度也不用设置为 size+1，改为 size 即可；“string2[size+1]= '\ 0'”改为“string2[size]= '\ 0'”。

对代码作以上修改，重新编译后，运行结果正常。输出如下结果：

```
The string is    Hello World!
The string printed backward is    !dlroW olleH 0
```

至此，我们完成了程序的调试工作。通过上面的例子，简单介绍了调试工具的基本使用方法，包括 gdb 程序的调用、在 gdb 中显示源文件、设置断点、观察变量、单步执行程序等。

2.9.2　gdb 的基本命令

gdb 是功能强大的调试器，支持的调试命令非常丰富，可以实现不同的功能。这些命令包括从简单的文件装入到允许检查所调用的堆栈内容的复杂命令。表 2-18 列出了使用 gdb 调试时会用到的一些命令。如果想了解 gdb 的详细使用方法，可以参考 gdb 的帮助文档。

表 2-18　gdb 的基本命令

命　　令	说　　明
file	装入想要调试的可执行文件
kill	终止正在调试的程序
list	列出产生执行文件的源代码的一部分
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
quit	退出 gdb
watch	动态监视一个变量的值
make	不退出 gdb 而重新产生可执行文件
call name(args)	调用并执行名为 name，参数为 args 的函数
return value	停止执行当前函数，并将 value 返回给调用者
break	在代码里设置断点，使程序执行到此处被挂起

2.9.3　gdb 的调用

一般情况下，调用 gdb 命令只使用一个参数：

```
gdb <可执行程序名>
```

同时，如果程序运行时产生了段错误，会在当前目录下产生核心内存映像 core 文件，可以在指定执行文件的同时为可执行程序指定一个 core 文件：

```
gdb <可执行文件名> core
```

除此之外，还可以为要执行的文件指定一个进程号：

```
gdb <可执行文件名> <进程号>
```

如下例所示，可以为 gdbtest 指定进程号 3000：

```
lxy@lxy-desktop:~/gdbtest$ gdb gdbtest 3000
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.   Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to program: /home/lxy/gdbtest/gdbtest, process 3000
ptrace: No such process.
/home/lxy/gdbtest/3000: No such file or directory.
(gdb)
```

首先，gdb 会寻找一个文件名为 3000 的文件，如果找不到，则把调试程序 gdbtest 的进程号(PID)设成 3000。

当 gdb 运行时，把任何一个不带选项前缀的参数都作为一个可执行文件、core 文件或要和被调试的程序相关联的进程号。不带任何选项前缀的参数和前面加了-se 或-c 选项的参数效果一样。gdb 把第一个前面没有选项说明的参数看作前面加了-se 选项，也就是需要调试的可执行文件并从此文件里读取符号表，如果有第二个前面没有选项说明的参数，将被看作是跟在-c 选项后面，也就是需要调试的 core 文件名。

如果不希望看到 gdb 开始的提示信息，可以用 gdb-silent 执行调试工作，通过更多的选项，开发者可以按自己的喜好定制 gdb 的行为。

输入 gdb-help 或-h 可以得到 gdb 启动时的所有选项提示。gdb 命令行中的所有参数都被按照排列的顺序传给 gdb，除非使用了-x 参数。

gdb 的许多选项都可以用缩写形式代表，这可以用-h 查看。在 gdb 中也可以采取任意长度的字符串代表选项，只要保证 gdb 能唯一地识别此参数就行。

表 2-19 列出了 gdb 一些最常用的参数选项。

表 2-19 gdb 常用的参数选项

选 项	说 明
-s filename	从 filename 指定的文件中读取要调试的程序的符号表
-e filename	在合适的时候执行 filename 指定的文件，并通过与 core 文件作比较来检查正确的数据
-se filename	从 filename 中读取符号表并作为可执行文件进行调试
-c filename	把 filename 指定的文件作为一个 core 文件
-c num	把数字 num 作为进程号和调试的程序进行关联，与 attach 命令相似
-command filename	按照 filename 指定的文件中的命令执行 gdb 命令，在 filename 指定的文件中存放着一系列的 gdb 命令，就像一个批处理
-d path	指定源文件的路径。把 path 加入到搜索源文件的路径中
-r	从符号文件中一次读取整个符号表，而不是使用默认的方式首先调入一部分符号，当需要时再读入其他一部分。这会使 gdb 的启动较慢，但可以加快以后的调试速度

2.9.4 gdb 运行模式的选择

可以用多种类模式来运行 gdb，例如采用“批模式”或“安静模式”。这些模式都是 gdb 运行时在命令行中通过选项来指定的。

表 2-20 列出了 gdb 运行模式的相关选项。

表 2-20 gdb 运行模式选项

选 项	说 明
-n	不执行任何初始化文件中的命令(一级初始化文件叫做.gdbinit)。一般情况下在这些文件中的命令会在所有的命令行参数都被传给 gdb 后执行
-q	设定 gdb 的运行模式为“安静模式”，可以不输出介绍和版权信息。这些信息在“批模式”中也不会显示
-batch	设定 gdb 的运行模式为“批模式”。gdb 在“批模式”下运行时，会执行命令文件中的所有命令，当所有命令都被成功地执行后 gdb 返回状态 0，如果在执行过程中出错，gdb 返回一个非零值
-cd dir	把 dir 作为 gdb 的工作目录，而非当前目录(一般 gdb 默认把当前目录作为工作目录)

2.10 小　　结

本章主要介绍了 Linux 下编程需要的一些基础知识，先介绍了编辑器的使用，然后介绍了 Linux 下最常用的 GNU 编译器 gcc 的调用格式、一般参数等。接着介绍了 Linux 下 make 工具的使用知识以及 autoconf 和 automake 程序维护工具的使用方法，最后介绍了 Linux 的调试工具 gdb 的基本使用方法。通过本章的学习，读者应对 Linux 下的编程环境有一定的了解，为后面 C 语言编程的学习扫清障碍。

习　　题

一、填空题

1. Linux 编程可分为______编程和______编程。
2. Linux 系统提供了许多文本编辑程序，比较常用的有______和______等。
3. 要使用 make，必须编写一个叫______的文件。
4. ______是一个用于生成可以自动配置软件源代码包以适应多种类 Unix 系统的 shell 脚本的工具
5. ______是一个从文件 Makefile.am 中自动生成 Makefile.in 文件的工具。

二、选择题

1. 用 gcc 编译 C 程序生成可执行文件的过程中，看起来像是一步就完成了，但实际上它要经历如下的四个步骤______。

   (A)　预处理、编译、汇编、链接　(B)　预处理、汇编、编译、链接
   (C)　链接、预处理、编译、汇编　(D)　编译、预处理、汇编、链接

2. 在 Makefile 文件中，使用变量的值方法是______。

   (A) $变量名　(B) $(变量名)　(C) #变量名　(D) #(变量名)

3. 由 autoconf 生成的脚本通常被命名为______。

   (A) configure　(B) gcc　(C) makefile　(D) make

4. automake 首先读入______文件，然后生成______。

   (A) Makefile.am、Makefile.in　(B) Makefile.in Makefile.am
   (C) Makefile.am、Makefile　(D) Makefile、Makefile.in

5. Linux 包含了一个______的调试程序，______是一个用来调试 C 和 C++程序的强力调试器，它使用户能在程序运行时观察程序的内部结构和内存的使用情况。

(A) gcc　(B) make　(C) gdb　(D) autoconf

三、上机题

1. 上机练习 vi、Emacs 的使用。
2. 在 Linux 上安装 gcc 开发工具，安装完成后，查看一下 gcc 的版本号。
3. 练习编写 Makefile 文件。
4. 按照书中给出的例子，练习 gdb 调试器的使用。

CHAPTER 3

Linux 下的文件编程

文件系统是现代操作系统中重要的组成部分之一。文件系统是指按一定规律组织起来的有序的文件组织结构，是构成系统中所有数据的基础。系统中的所有文件都驻留在文件系统中某一特定的位置。Linux 系统提供的文件系统是树形的层次结构系统。所有文件最终都归结到根目录“/”。Linux 支持多种文件系统，在此不对其进行详细叙述。当前最通用的文件系统是 ext2 系统。用户也可以根据需要自行选取。

每种文件系统类型存储数据的基本格式都是不一样的。但是，在 Linux 下访问任何文件系统时，系统都把数据整理成一个目录树下的文件，并包括我们熟悉的文件的属主和组 ID、保护位以及其他特征。事实上，属主、保护等信息只有那些能存储 Linux 文件的文件系统类型才能提供。对于没有存储这些信息的文件系统类型，用来访问这些文件系统的内核驱动程序会“伪造”这些信息。例如，MS-DOS 文件系统没有文件属主的概念，但所有文件都显示成属主是 root。用这种方法，在一定层次上，所有文件系统都很相似，每个文件都有一定的属性。至于这些文件属性是否真的在文件系统底层被使用，就是另外一回事了。

前面已经介绍过，Linux 同 Unix 系统一样，将目录和设备当作特殊文件来处理。这种处理方法使所有与文件相关的系统调用，无论对字符设备的操作还是对块设备的操作，从程序设计人员的角度来看是完全一样的，因为其接口非常一致，所以使用起来十分简便。本章介绍 Linux 系统中的文件以及与文件有关的操作。在 C 编程环境中，与文件有关的操作主要是 I/O 操作，即基于文件描述符的 I/O 操作。此外，还将介绍其他一些与文件有关的操作。

3.1 概　　述

与其他操作系统相比，Linux 的文件系统更为简单统一。Linux 的文件是个简单的字节

序列。因此，一个文本文件(由 ASCII 字符组成的字符流)和一个二进制文件的结构和访问方法对于 Linux 是一样的，差别仅在文件内容本身，这要由用户程序来解释。

文件是由一系列块(block)组成，每个块可能含有 512、1024、2048 或 4096 个字节，具体由系统实现决定。不同的文件系统的块大小可以不同，但同一个文件系统的块大小是相同的。使用的块较大时，由于每次磁盘操作可以传输更多的数据，操作所花的时间较少，所以可以提高磁盘和内存间数据的传输率；但与此同时，块太大时，存储的有效容量也将会下降。

Linux 的文件系统通常由 4 部分组成：引导块、超级块、索引节点表(inode table)和数据块。其中：

- 引导块用来存放文件系统的引导程序，用于系统引导或启动操作系统。如果一个文件系统不安放操作系统，它的引导块将为空。
- 超级块用来描述本文件系统管理的资源，它包含空闲索引节点表和空闲数据块表，具体说明文件系统的资源使用情况。
- 索引节点表用来存储文件的控制信息，每个节点对应一个文件。
- 数据块是磁盘上存放数据的磁盘块，包括目录文件和数据。

Linux 的文件系统用图来表示就是如图 3-1 所示。

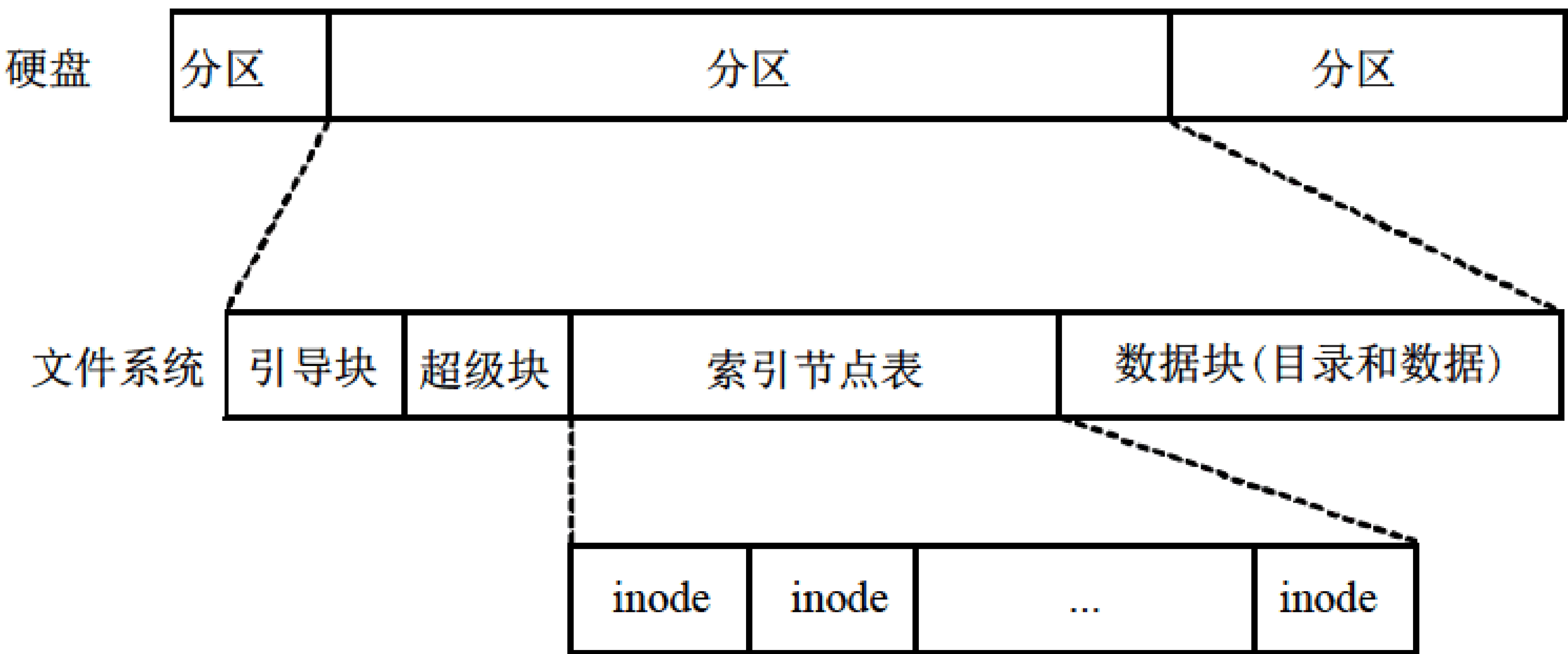


图 3-1　Linux 文件系统

3.1.1　超级块

超级块用于描述一个文件系统的资源状态，例如：文件系统的大小、空闲空间位置信息。超级块由如下字段构成：

- 文件系统的规模(如 inode 数目、数据块数目、保留块数目和块的大小等)。
- 文件系统中空闲块的数目。
- 文件系统中部分可用的空闲块表。
- 空闲块表中下一个空闲块号。
- 索引节点表的大小。
- 文件系统中空闲索引节点表数目。

- 文件系统中部分空闲索引节点表。
- 空闲索引节点表中下一个空闲索引节点号。
- 超级块的锁字段。
- 空闲块表的锁字段和空闲索引节点的锁字段。
- 超级块是否被修改的标志。
- 其他字段。

从上述可以看出，超级块在文件系统存取和管理文件上起着至关重要的作用。其中，锁字段是为了保证互斥操作。在其他字段中标出该文件系统是否完整。如 Linux 在关机时要求先将缓冲区数据写回文件系统，并拆卸(umount)该文件系统，如果没有拆卸文件系统就关机，很可能导致数据丢失，所以在安装一个文件系统时会坚持超级块中的字段，如果上次没有做拆卸，就要对文件系统完整性做检查(fsck)。

3.1.2 索引节点(inode)

索引节点(inode)是 Linux 文件系统最基本的概念。一个文件的控制信息通常由 inode 给出，每个 inode 对应着一个文件。在 inode 中包含有文件数据在磁盘上存储的位置信息，还包含有存取权限、文件所有者及存取时间等信息。索引节点存储在磁盘上，内核把 inode 读进内存索引节点来操作它和它所对应的文件。为了便于理解，我们把存储在磁盘上的 inode 称作磁盘索引节点，而把它在内存中的映像称作内存索引节点。

磁盘索引节点由如下字段构成。

- 文件类型：文件可以是普通文件、目录文件、链接文件、设备文件、管道文件。
- 文件链接数：记录了引用该文件的目录表项数，即记录了有多少个文件名指向该文件。
- 文件属主标识：指出该文件的所有者 id。
- 文件属主的组标识：指出该文件所有者属组的 id。
- 文件的访问权限：系统将用户分为文件属主、同组用户和其他用户三类，每类用户可能获得对文件一种或几种访问权限。要特别指出的是，目录文件的执行权限是指修改目录的权力。
- 文件的存取时间：包括文件最后一次被修改的时间、最后一次被访问的时间和最后一次修改索引节点的时间。
- 文件的长度：以字节表示的文件长度。
- 文件的数据块指针：对文件操作的当前位置指针。

索引节点中并不包含文件名，文件名信息存放在目录文件中。在系统中定义了 stat 结构(该结构定义于/usr/include/sys/stat.h 文件中)来存放这些信息。

```
struct stat
   {
```

```
    __dev_t       st_dev;      /*文件所在设备号，高字节为主设备号，低字节为从设备号*/
    __ino_t       st_ino;      /* 文件的索引节点号*/
    __mode_t      st_mode;     /* 文件模式 */
    __nlink_t     st_nlink;    /*与该文件硬链接的数量*/
    __uid_t       st_uid;      /* 文件属主用户 ID*/
    __gid_t       st_gid;      /* 文件属主所在组 ID*/
    __dev_t       st_rdev;     /* 如果是一个设备文件则指出其所代表的设备号，对其他类型
                               的文件无意义 */
    __off_t       st_size;     /* 以字节计算的文件长度，对于设备文件此项为 0*/
    __blksize_t   st_blksize;  /* 文件系统 I/O 的块尺寸*/
    __blkcnt_t    st_blocks;   /* 分配的块数*/
    __time_t      st_atime;    /* 最近一次的访问时间 */
    __time_t      st_mtime;    /*最近一次修改时间*/
    __time_t      st_ctime;    /* 最近一次状态改变时间*/
};
```

实际的 stat 结构还要复杂一些，在此只引用了其中比较重要的部分。

3.1.3 文件类型

在此再具体介绍一下 Linux 系统中不同类型的文件。

3.1.3.1 普通文件

普通文件也称正规文件，是最常见的一类文件，也是最常使用到的一类文件。其特点是不包含有文件系统的结构信息。通常所接触到的图形文件、数据文件、文档文件、声音文件等都属于普通文件。这种类型的文件按其内部结构又可细分为两个文件类型：文本文件和二进制文件。

- 文本文件：文本文件是以字符(通常是 ASCII 码)表示的，是以行为基本结构的信息存储方式。
- 二进制文件：二进制文件是按信息在内存中的格式表示的，它通常不能直接查看，而必须使用相应的软件。

3.1.3.2 目录文件

Linux 文件系统的目录是一种文件，在文件名与索引节点之间的转换起到桥梁作用，是树形文件结构的关键。Linux 的目录文件其实非常简单，它的主要内容只有两项：文件名和索引节点号，如图 3-2 所示。

索引节点号	文件名

图 3-2　Linux 的目录文件

Linux 的文件系统对文件管理是通过索引节点来进行的，目录文件只不过提供了文件名和索引节点之间的转换手段。为了保证文件系统层次的完整性，目录文件是由系统来管理的，用户只能读目录文件，而不允许直接写目录文件。每个目录文件的前两项是两个特设的文件“.”和“..”。其中“.”对应于该目录文件本身的索引节点，而“..”则对应于其父目录的索引节点。如果一个目录中只包含“.”和“..”文件，则该目录为空目录。

当用户访问某个文件时，系统需要找到它所对应的索引节点。目录文件建立了文件名和索引节点号之间路径的路线。比如，考虑路径“../a/b”，它要从当前目录开始，到达其父目录，再到达其父目录的子目录 a，访问那个目录中的名为 b 的文件。为此，系统要完成如下步骤：

(1) 检索当前目录的索引节点。

(2) 通过当前目录的索引节点，找到当前文件，查出父目录“..”。

(3) 检索“..”的索引节点。

(4) 通过父目录“..”的索引节点，检索父目录文件，查出文件“a”的索引节点号。

(5) 检索“a”的索引节点。

(6) 利用“a”的目录索引节点中的信息，检索“a”目录文件，查到“b”的索引节点号。

(7) 检索文件“b”的索引号。

(8) 访问文件“b”。

以上步骤看似复杂，实际上内存中有索引节点表，大多数情况下检索动作主要在内存中即可完成。

3.1.3.3 链接文件

链接文件是一种特殊的文件。它实际上是指向一个真实存在的文件的链接。比如用户要在一个目录文件中使用其他目录文件下的文件时，并不需要将其复制过来，而只需在此目录中建立一个链接文件指向所要调用的文件。在具体使用时，并不会感觉到它们有什么不同。根据链接对象的不同，链接文件又可以细分为硬链接文件和符号链接文件。

3.1.3.4 设备文件

设备文件是 Linux 中最特殊的文件。正是由于它的存在，使得 Linux 系统可以十分方便地访问外部设备。Linux 系统为外部设备提供一种标准接口，将外部设备视为一种特殊的文件。用户可以像访问普通文件一样访问外部设备。这就使 Linux 系统可以很方便地适应不断发展的外部设备。通常 Linux 系统设备文件放在/dev 目录下。设备文件使用设备的主设备号和次设备号来指定某外部设备。主设备号用于说明设备类型，次设备号用于说明具体设备。例如，以 IDE 硬盘为第一主盘，它的第三个分区的设备文件就是/dev/hda3。其中 hd 是主设备号，a3 是次设备号。根据访问数据方式的不同，设备文件又可以细分为两种类型：块设备文件和字符设备文件。

- 块设备文件：块设备文件是以固定长度的块访问数据的。

● 字符设备文件：字符设备文件是以指定字符(通常是一个字符)访问数据的。

大多数外部设备都提供两种访问方式。但对每一种设备来说，都有其最佳的访问方式。

3.1.3.5 管道文件

管道文件也是一种很特殊的文件。主要用于不同进程间的信息传递。当两个进程间需要进行数据或信息传递时，可以通过管道文件。一个进程将需传递的数据或信息写入管道的一端，另一进程则从管道的另一端取得所需的数据或信息。通常管道是建立在高速缓存中的。采用先进先出的规定处理其中的数据。可以细分为有名管道和无名管道两种。

3.2 文件描述符

C 语言的标准输入输出库中的库函数，如 fopen，fclose，fread，fwrite 等，提供的是高层服务；而 Linux 的文件 I/O 调用提供的是底层的服务，底层的服务不提供缓冲而直接进入操作系统。标准输入输出库中的高层服务归根到底还是要调用 Linux 所提供的底层服务。

在介绍具体的调用之前，我们先了解一下文件描述符。

对于 Linux 而言，所有对设备和文件的操作都使用文件描述符来进行。文件描述符是一个非负的整数，表示为 int 类型的对象，它是一个索引值，并指向内核中每个进程打开文件的记录表。当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符。当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。

每个进程都可以拥有若干文件描述符，数量多少则依赖于操作系统的实现，Linux 中的每个进程可以有 1024 个文件描述符。每个进程有自己的用户描述符表。文件描述符表的前三项对于一般的进程是固定的且是由系统自动打开的。文件描述符 0 是标准输入文件，对于一般进程来说是键盘；文件描述符 1 是标准输出文件，一般是输出到显示器；文件描述符 2 是标准错误输出文件，一般也是输出到屏幕。用户程序不用执行打开操作就可直接使用。文件描述符 0、1、2 对应的符号常量分别是 STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO，它们都定义在头文件<unistd.h>中。

3.3 基本文件 I/O 操作

本节将详细讲述普通文件的基本输入和输出操作，包括建立文件(create)、打开文件(open)、文件的写操作(write)、文件的读操作(read)、关闭一个打开的文件(close)、设置文件读写指针位置(1seek)等。

3.3.1 open 函数

调用 open 函数可以打开或创建一个文件，open 是进程存取一个文件中的数据必须首先完成的系统调用。open 函数的格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname,int flags,… /* mode_t mode */);
```

将第三个参数写为“…”，ISO C 用这种方法表明余下参数及其类型根据具体的调用会有所不同。对于 open 函数而言，仅当创建新文件时才使用第三个参数。在函数原型中将此参数放置在注释中。

open 函数打开一个文件并返回一个文件描述符。open 函数中的第 1 个参数 pathname 是要打开(或要创建)的文件名或含路径的文件名。第 2 个参数 flags 是标志打开的方式，可以是 O_RDONLY 表示请求以只读方式打开文件，O_WRONLY 表示请求以只写的方式打开文件，O_RDWR 表示以可读写的方式打开文件。对一个已经存在的文件，在打开它时还要注意它的存取权限。假设用户 user 是 file1 文件的拥有者，file1 文件的存取权限被设为 -r-xr—r--，这表明 user 只有读和执行的权力，如果用户 user 要对文件 file1 做 open("file1",O_RDWR)操作时，会返回错误，因为用户没有写该文件的权限。

标志 O_RDONLY、O_WRONLY、O_RDWR 的取值分别为 0、1、2，它们之间不能用 OR 的方法联合使用。表 3-1 列出了其他可用的选项。

表 3-1 flags 的其他选项

选 项	说 明
O_CREAT	当文件不存在时，将建立该文件，此时会用到 open 的第三个参数
O_EXCL	如果同时指定了 O_CREAT，而文件已经存在，则会出错。用此可以测试一个文件是否存在，如果不存在，则创建此文件
O_NOCTTY	当文件名(可以包含路径，即第一个参数 pathname)指向一个终端设备，它将不再是进程控制的终端，即使该进程没有一个终端设备
O_TRUNC	如果文件存在，则该文件将被截断，即长度截断为 0。注意，文件没有以写方式打开也可以截断；截断后文件的属主和属性不变
O_APPEND	文件以追加方式打开 i，每次进行写操作时，文件指针都会被放置到文件末尾
O_NONBLOCK 或 O_NDELAY	当文件以非阻塞方式打开后，对于 open 及随后的对该文件的操作，都会及时返回，而无须进程等待。这对于普通文件和目录文件没有作用，但对于管道等进程间通信的操作很有用
O_SYNC	文件以同步 I/O 方式打开。任何写操作都会使得进程被阻塞，直到物理写动作完成为止

上述标志可以混合使用，各标志之间用“|”符号连接。其实第二个参数为 int 型参数，该数的每位都对应一个操作，符号“|”是将它们按位或，使得需要操作的位被置 1。例如需要以“可读写、如果文件存在就截断、不存在就创建”的方式打开一个位于/home/test 目录下的名为 file1 的文件，可以用下面的调用实现：

```
int fd;
fd=open("/home/test/file1",O_RDWR|O_CREAT|O_TRUNC,mode);
```

上面介绍的标志中有一些可以在文件打开后用 fcntl 改变，详细情况见 fcntl 函数一节。open 函数第 3 个参数是在 O_CREAT 创建文件时使用，指出新建文件的存取权限。第 3 个参数的符号常数如表 3-2 所示，表中的值为 8 进制。

表 3-2　open 函数的第 3 个参数的符号常数

符　　号	值	含　　义
S_IRWXU	00700	文件属主有读、写、执行的权限
S_IRUSR(S_IREAD)	00400	文件属主有读权限
S_IWUSR(S_IWRITE)	00200	文件属主有写权限
S_XWUSR(S_IEXEC)	00100	文件属主有执行权限
S_IRWXG	00070	文件组成员有读、写、执行权限
S_IRGRP	00040	文件组成员有读权限
S_IXGRP	00010	文件组成员有执行权限
S_IRWXO	00007	其他用户有读、写、执行的权限
S_IROTH	00004	其他用户有读权限
S_IWOTH	00002	其他用户有写权限
S_IXOTH	00001	其他用户有执行权限

上面给出了第三个参数 mode 的符号常数，可以直接使用它们，也可以进行多个运算后得到 mode 值(一般使用按位或的运算来增加权限)，上面也给出了符号对应的值，读者可以自行计算。

open 函数返回一个文件描述符，在随后对文件的所有操作中将使用这个文件描述符。当打开一个文件时，Linux 总是分配最低编号的空闲文件描述符。文件描述符在实现输入/输出、重新定向到文件和管道时，将得到很好的利用。open 函数打开失败时返回-1。

3.3.2　close 函数

close 函数用来关闭不再使用的文件，close 函数的格式如下：

```
#include <unistd.h>
```

```
int close(int fd);
```

其中，fd 是文件描述符，它可以通过 open、pipe、dup 等函数取得。close 如果执行成功，返回值为 0，否则返回值为-1。

3.3.3　read 函数

read 函数是从文件中读取指定长度的数据到内存中，其格式如下：

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

read 函数的第 1 个参数是文件描述符，第 2 个参数是输入缓冲区指针，第 3 个参数是要读入的字节数，数据类型 size_t 为 unsigned int，是一个类型别名。read 函数的功能是从文件描述符所指定的文件中读取 count 个字节到 buf 所指向的内存缓冲区中。如果 count 参数为 0，该系统调用返回 0 并且没有其他结果。如果 count 大于 SSIZE_MAX，则结果不能确定。这里提到的 SSIZE_MAX 在 POSIX 中是 32767。

当 read 函数执行成功时，该函数的返回值是读取的字节数，返回 0 表示文件指针在文件尾。成功读取一定字符数返回时，文件指针也向后移动一定的字符数。当返回值小于请求的字符数 count 时，并不意味着产生了错误，出现这种情况可能是因为：已经接近文件尾，没有 count 这么多个字节可读；或者进程正在从管道或终端读取数据；或者 read 函数被某个信号(signal)中断。当 read 函数产生错误时，返回值为-1。

ssize_t 与 size_t 类似，Linux 系统用它表示在一次操作中可以读写的块大小。在 32 位的系统中，它是一个 32 位的整数类型；在 64 位的系统中是 64 位的长整类型。不过与 size_t 不同的是，它是一个有符号整数类型以便能够表示错误情形的返回值。read 返回的实际读入的字节数在以下几种情况下可能小于 count：

- 当读的是普通文件且在还未读够所请求的字节数便遇到了文件尾时。例如，如果在文件中只剩下 30 个字节便到达文件尾，而却企图读 100 个字节时，read 将返回 30。下一次再调用 read，它将返回 0，指出文件结束。
- 当所读的文件对应于终端设备时，通常每次至多只读一行。
- 当从网络读入时，网络的内部缓冲可能导致读入的字节数少于所请求的字节数。
- 某些面向记录的设备，例如磁带，每次至多只读一个记录。
- 当成功地读出一些数据之后被一信号中断时，可能返回读出的字节数，也可能返回-1。

对于普通文件或其他可以定位的文件，read 从文件的当前位置开始读数据；在成功返回之前，文件位置增加实际读的字节数。对于不能定位的文件，read 从文件的当前位置读数据。

3.3.4 write 函数

write 函数是将内存中的数据写入文件，其声明格式如下：

```
#include <unistd.h>
ssize_t read(int fd, const void *buf, size_t count);
```

write 函数的第 1 个参数是文件描述符，第 2 个参数是输出缓冲区地址指针，第 3 个参数是要写入的字节数，它的功能是将 buf 所指内存中的 count 个字节写入文件描述符 fd 所指的文件。POSIX 要求在一个 write 之后的 read 函数应该返回新的数据。调用成功时，write 调用返回写入的字节数(为 0 表示没有数据要写)。当发生错误时，调用返回-1。如果 write 调用的参数 count 为 0，且文件描述符 fd 指向一个普通文件，则调用将返回 0 且没有其他任何实际影响。

3.3.5 creat 函数

creat 是进程新建一个文件时使用的函数。新建文件的功能也可以由 open 函数实现。creat 函数的声明格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

此函数等效于：

```
int open(const char *pathname, O_WRONLY|O_CREAT | O_TRUNC, mode_t mode);
```

creat 函数中的参数 pathname 和 mode 的含义与函数 open 中的一样。如果 pathname 指向的文件不存在，系统就以指定的文件名和权限创建一个新文件；如果 pathname 指向的文件存在，系统就将该文件截断，释放以前数据所占用的磁盘块。对文件截断的操作受原文件存取权限的限制。creat 函数中的参数 mode 与 umask 计算生成存取权限的方法与 open 一样。这里要注意的是，open 可以打开一个特殊设备文件，而 creat 不能创建设备文件，创建特殊文件要用函数 mknod 来代替。

当调用成功时，creat 函数返回值为该文件的描述符。此时文件以只读方式打开。失败时返回值为-1。

下面利用前面介绍的函数编写一个简易的拷贝程序。

例 3-1 简易拷贝程序

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
```

```
3    #include <fcntl.h>
4    #include <unistd.h>
5
6    #define PERMS 0666
7    #define     DUMMY 0
8    #define BUFSIZE 1024
9    main(int argc, char *argv[])
10   {
11        int source_fd, target_fd, num;
12        char iobuffer[BUFSIZE];
13        if(argc!=3)
14        {
15          printf("Usage: copy Sourcefile Targetfile\n");
16          return 1;
17        }
18        if((source_fd=open(*(argv+1),O_RDONLY,DUMMY))==-1)
19        {
20          printf("Source file open error!\n");
21          return 2;
22        }
23        if((target_fd=open(*(argv+2), O_WRONLY|O_CREAT, PERMS))==-1)
24        {
25          printf("Target file open error!\n");
26          return 3;
27        }
28
29        while((num=read(source_fd, iobuffer, BUFSIZE))>0)
30        if(write(target_fd, iobuffer, num)!=num)
31        {
32          printf("Target file write   error!\n");
33          return 4;
34        }
35        close(source_fd);
36        close(target_fd);
37        return 0;
38   }
```

编译上述代码时，指定编译后输出的可执行文件名为 copy。

在本例之前的程序中，我们使用的 main 函数都是不带参数的。因此 main 后的括号都是空括号。实际上，main 函数可以带参数，这个参数可以认为是 main 函数的形式参数。C 语言规定 main 函数的参数只能有两个，习惯上这两个参数写为 argc 和 argv。因此，main 函数的函数头可写为：

```
main (argc, argv)
```

C 语言还规定 argc(第一个形参)必须是整型变量，argv(第二个形参)必须是指向字符串的指针数组。加上形参说明后，main 函数的函数头应写为：

```
main(int argc, char *argv[])
```

由于 main 函数不能被其他函数调用， 因此不可能在程序内部取得实际值。那么，在何处把实参值赋予 main 函数的形参呢？实际上，main 函数的参数值是从操作系统命令行上获得的。当运行一个可执行文件时，在 Linux 命令行下键入文件名，再输入实际参数即可把这些实参传送到 main 的形参中去。

命令行的一般形式为：

```
可执行文件名 参数 参数…;
```

应该特别注意的是，main 的两个形参和命令行中的参数在位置上不是一一对应的。因为，main 的形参只有二个，而命令行中的参数个数原则上未加限制。argc 参数表示了命令行中参数的个数(注意：可执行文件名本身也算一个参数)，argc 的值是在输入命令行时由系统按实际参数的个数自动赋予的。例如有命令行为：

```
gcc –o a.out a.c
```

由于文件名 gcc 本身也算一个参数，所以共有 4 个参数，因此 argc 取得的值为 4。argv 参数是字符串指针数组，其各元素值为命令行中各字符串(参数均按字符串处理)的首地址。指针数组的长度即为参数个数。数组元素初值由系统自动赋予。在上面命令中，agv 数组的第 1 个元素指向的字符串为“gcc”，第 2 个元素指向的字符串为“-o”，第 3 个元素指向的字符串为“a.out”，第 4 个元素指向的字符串为“a.c”。

此外，main 函数也带有返回值，默认的返回值类型为 int，在一般的程序中， main 函数的返回值类型 int 可以省略不写。返回值传递给程序的激活者(如操作系统)。如果 main 函数的最后没有写 return 语句的话，gcc 会自动在生成的目标文件中加入

```
return 0;
```

表示程序正常退出。

在例 3-1 中，首先根据命令行参数个数判断命令输入是否正确，如果输入不正确，输出命令使用方法的提示信息后，程序退出(第 13~17 行)。如果命令参数个数符合要求，则根据输入的参数打开原始文件，如果打开错误，输出错误信息后，程序退出(第 18~22 行)。然后再打开目标文件，如果目标文件不存在，就建立它。如果建立目标文件不成功，程序输出错误提示信息后，退出(第 23~27 行)。打开目标文件成功后，接下来进行文件读取与写入的工作。每次从原始文件中读取大小为 BUFSIZE 的内容到 iobuffer 中，然后将 iobuffer 中的内容写到目标文件中。如果写入的过程中失败，输出错误提示信息后，程序退出。全部内容复制完成后，关闭原始文件和目标文件，程序正常退出(第 29~37 行)。

3.3.6 lseek 函数

每个打开的文件都有一个与其相关联的当前文件偏移量(也叫文件指针)，它通常是一个非负整数，用以度量从文件开始处计算的字节数。通常，读、写操作都从当前文件偏移量处开始，并使偏移量增加所读写的字节数。进程可以使用 lseek 函数来指定文件偏移量的位置，从而实现文件的随机存取。其声明格式如下：

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fds, off_t offset, int whence);
```

lseek 函数的第 1 个参数是文件描述符，第 2 个参数是偏移量，指的是每一读写操作所需移动距离，以字节数量作单位，这个值可正可负。正值指的是向前移，负值指的是向后移。第 3 个参数 whence 是当前位置的基点，它的取值如表 3-3 所示。

表 3-3 whence 的取值选项

选　项	说　明
SEEK_SET	当前位置为文件的开头，新位置位偏移量的大小
SEEK_CUR	当前位置为文件指针的位置，新位置为当前位置加上偏移量
SEEK_END	当前位置为文件的末尾，新位置为文件的大小加上偏移量的大小

lseek 函数允许文件偏移量被设置到超过文件结束符(EOF)处。如果这样的话，下一次调用 write 时，可以将文件的长度延伸到所需的长度，并用无意义的字符填充这个空隙。如果随后的 read 读取这个空隙间的数据，将得到无意义的值，直到这个文件数据块被真正写回到磁盘上，再读取这个空隙间的数据将得到 0。这是因为，当在文件尾之后执行 write 函数时，Linux 系统并不存储无用的数据块。在 read 函数读到该数据块时，系统为 read 函数产生一个全为 0 的数据块，返回给 read 函数。如果一个 read 函数置于文件尾或文件尾之后的文件偏移量，则产生 0 作为 read 的返回值。

当 lseek 调用成功时，返回值为一个以字节为单位从文件头开始计算文件偏移量的值。调用失败时，返回值为-1。

例 3-2 下面的示例结合了前述的几个系统函数，程序建立了一个简单的文本文件，文件里每批数据都是以结构的方式存取，每一个结构代表的是数据文件中的一批记录。

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3    #include <stdio.h>
4    #include <fcntl.h>
5    #include <unistd.h>
6
```

```
7     #define LNAME 9
8     #define    PERMS 0666          /* 新建文件的权限 */
9     #define DATAFILE "datafile"        /* 要写入数据的文件名字 */
10    #define    USERS 10
11
12    typedef struct user{
13        int          uid;
14        char    login[LNAME+1];
15    } RECORD;
16
17    char *user_name[]={"u1","u2","u3","u4","u5","u6","u7","u8","u9","admin"};
18    main()
19    {
20        int      i, fd;
21        RECORD    rec;
22        if((fd=open(DATAFILE,O_WRONLY|O_TRUNC|O_CREAT,PERMS))==-1)
23        {
24        perror("open");
25        return 1;
26        }
27    for(i=USERS-1;i>=0;i--)
28    {
29        rec.uid=i;
30        strcpy(rec.login,user_name[i]);
31        lseek(fd,(long)i*sizeof(RECORD),SEEK_SET);
32        write(fd,(char *)&rec,sizeof(RECORD));
33    }
34    lseek(fd,0L,SEEK_END);
35    close(fd);
36    return 0;
37    }
```

这个程序很简单，首先用 open 函数建立一个文件并打开该文件，文件名为 datafile(第 22 行)。如果文件建立不成功，则用标准库函数 perror 来显示错误信息(第 24 行)，它的原型如下：

```
#include <stdio.h>
void perror (const char *s);
#include <error.h>
const char * sys_errlist[];
int sys_nerr;
```

perror 用来将上一个函数发生错误的原因输出到标准错误(stderr)。参数 s 所

指的字符串会先打印输出，后面再加上错误原因字符串。此错误原因依照全局变量 errno 的值来决定要输出的字符串。C 标准函数库内，提供一个叫 sys_errlist 的全局数组，里面存储着各类系统错误的信息，而 errno 则是 sys_errlist 的下标。另外还有一个外部变量叫 sys_nerr，它是用来记录系统错误信息的总数。

在输出错误信息之后，程序返回。如果文件成功建立并打开，则进入循环体。每次循环首先给结构 rec 的 uid 赋值，然后把字符数组里的元素通过 strcpy 函数复制到结构 rec 的 login_user 中。然后通过 lseek 设定文件偏移量，用 write 函数把 rec 的值写入文件中(第 27~33 行)。在循环完成后，又调用 lseek 函数把文件偏移量设定在最后，随后调用 close 函数关闭文件，并退出程序(第 34~36 行)。

3.4 文件高级操作

本节在上一节的基础上继续介绍有关文件操作的一些高级功能，包括文件的保护、文件的操作和文件状态信息的获取等。

3.4.1 文件模式

前面提到设置文件的存取权限，分为属主、同组用户和其他用户三类。每类分为读、写和执行权限。其实这只是文件模式的一部分，属于文件模式的低 9 位(二进制)。下面再介绍一下文件模式的高 7 位(二进制)。表 3-4 指出了文件模式各位的含义，表中的值为 8 进制。

表 3-4 文件模式定义的常量符号

符 号	值	含 义
S_IFMT	0170000	这些位决定文件类型
S_IFDIR	0040000	目录文件
S_IFCHR	0020000	字符设备文件
S_IFBLK	0060000	块设备文件
S_IFREG	0100000	普通文件
S_IFIFO	0001000	有名管道文件
S_ISUID	0400000	Set UID 标志
S_ISGID	0200000	Set GID 标志
S_IFLNK	0120000	符号链接
S_IFSOCK	0140000	套接字(Socket)

表 3-5 中第一项 S_IFMT 是用来检测文件类型的屏蔽码。例如，某文件的模式为 mode，那么可用表达式(mode&S_IFMT)来检测文件的类型。如果表达式的值等于 S_IFDIR，则表示该文件为目录文件。因为在文件类型中，有些文件类型是由 2 位二进制位决定的，所以不能简单地判断一位的值。例如，一个文件是目录文件，如果用(S_IFDIR&mode)==S_IFDIR 并不能就此断定该文件是目录文件。所以应该用 S_IFMT 作文件检测模式的屏蔽码。如果读者不清楚系统都定义了哪些文件模式的常量符号，一个办法是直接用数值，另一个办法是查看 stat.h 文件(这些常量都定义在<sys/stat.h>中)。

另外需要注意的是，在函数 creat、open、chmod 和 mknod 中，文件模式的右边 12 位都可以使用，即除了存取权限外，还包括 S_ISUID、S_ISGID。而最左边的 4 位仅在 mknod 函数中使用。不过，普通文件位 S_IFREG 也可以由 creat 和 open 函数设置。

3.4.2　确定和改变文件模式

对文件的各种模式位有所了解之后，这一节介绍确定和改变文件访问权限的有关函数。

3.4.2.1　umask 函数

Linux 系统中每一个文件都是由进程(关于进程在后面的内容还要介绍)创建的，进程在调用 open 或 creat 创建文件时需指定文件的访问权限。同时，与每一个进程相连还有一个文件创建屏蔽，称为 umask。文件创建屏蔽与文件方式字一样是一个位串，并且与文件方式位一一对应。每当进程创建一个新文件或新目录时，它所指定的文件访问权限位将受到文件创建屏蔽 umask 的作用。这种作用正如 umask 的名字所隐喻，表示的是文件创建时受到“屏蔽”，也即遭到禁止的访问权限。例如，如果 umask 的值为 007(即其他用户的所有访问权限位为 1)，则表示新创建文件的其他用户访问权限位均为 0，即其他用户没有任何访问权限，即使创建函数中指定 mode 参数允许这种权限也不例外。

每一个进程从其父进程继承它的 umask 值，shell 进程的 umask 将传给它所执行的每一个进程。可以用 umask 命令查看和设置 shell 进程的 umask 之值：

```
$ umask
0002
```

这表示当前的 umask 仅屏蔽了其他人的写权限。如果只想让同组用户有读和执行权限，其他用户有执行权限，则可以如下设置 umask：

```
$ umask 026
```

一旦这个命令被执行，将来创建的任何文件都将受到它的保护。

如果我们想在创建一个新文件时保证指定的访问权限有效，则在进程运行时必须保证适当的 umask 值。例如，如果想保证让任何人能以任何方式访问一个文件，就应当设置 umask 为 0；否则的话，当进程运行时起作用的 umask 值会导致想要的权限位被清除。

改变进程文件创建屏蔽值的函数是 umask。

umask 函数的格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

其中，参数 cmask 是新设置的文件创建屏蔽码。uamsk 函数用 cmask 设置进程的当前文件创建屏蔽，然后返回 umask 原来的屏蔽值。cmask 参数由表 3-2 中 9 个权限位中的任何一个按位或运算而形成。

例如，当文件的创建屏蔽码为 042 时，用 creat("test",0777)创建的文件 test 的文件访问权限就不是 0777，而是 0735。

3.4.2.2 chmod 和 fchmod 函数

如前面所说，一般用户很少使用 umask 函数，事实上通常只有 shell 才用它来改变文件创建屏蔽，大多数用户常用的是 chmod 函数。

chmod 函数的格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char * filename, mode_t mode);
int fchmod(int fd, mode_t mode);
```

其中，第 1 个参数 filename 是被设置的文件名(包含路径)，第 2 个参数 mode 是文件模式。

chmod 函数用于修改任何类型的一个现存文件的存取权，调用成功时返回 0，否则返回-1。chmod 只对文件的 inode 节点进行操作，修改文件模式，而不修改文件本身。调用 chmod 的进程，它的有效用户标识必须是超级用户或文件属主的用户标识，否则调用失败；当对只读文件系统中的文件进行 chmod 操作时，调用也将失败。

fchmod 与 chmod 类似，不同的是它操作的是已打开文件描述符给出的文件。

如果进程的用户标识不是超级用户且欲设置文件的组标识与进程的有效组标识 ID(EGID)不匹配或与进程任何一个其他附属的组标识不匹配，那么 S_ISGID 位将被置 0。

3.4.2.3 chown 和 fchown 函数

Linux 中每个用户有一个唯一的账号，该账号具有一个唯一的注册名和一个唯一的用户 ID。每个用户可以是一个或多个组的成员，其中一个组是用户的初始组。/etc/passwd 文件中用户的登记项定义了用户的初始组，组文件/etc/group 则定义了用户的另外一些组，这些组在需要时也可以指定其他用户。例如，注册名是 lxy，如果在/etc/passwd 文件中组名是 users，并且在/etc/group 文件中还指定 lxy 是 group1 和 group2 组的成员，那么 lxy 就是 3 个组的成员，这 3 个组分别是：users、group1 和 group2。其中，users 是 lxy 的初始组，group1 和 group2 是 lxy 的二级组。

Linux 系统中用户和组关系总是以某种方式遗传给进程以及它们所创建的文件。系统根据用户和组关系来判定一个文件属于谁，以及进程是否有权进行某种操作或访问某个文件。

每个文件有一个属主和一个组。用 ls 命令可以查看文件的属主和属主所在的组，例如：

```
ls -l test.c
-rw-r--r-- 1 lxy users 86 2008-01-03 13:22 test.c
```

文件 test.c 的属主是 lxy，组是 users。文件的属主只要愿意就可以对文件进行任何操作，如读写、重新命名以及删除等。属于同一个组的用户可以共享该组的文件。

在程序中，文件属主用属主的用户 ID 来表示，结构 stat 的成员 st_uid 给出文件属主的用户 ID，成员 st_gid 则给出文件属主的组 ID。

每个进程有 6 个以上的 ID 与它相连。这些 ID 如表 3-5 所示。

表 3-5　与每个进程相连的各种用户 ID 和组 ID

ID	说明
实际用户 ID 实际组 ID	指明实际用户是谁
有效用户 ID 有效组 ID 附加组 ID	用于文件的访问权限测试
保存的调整用户 ID 保存的调整组 ID	由 exec 保存

在表 3-6 所列的 ID 中：

- 实际用户 ID 和实际组 ID 标识实际用户是谁。这两个 ID 值取自注册口令文件/etc/passwd 中的登记项，它们反映的是执行进程的真实用户。
- 有效用户 ID、有效组 ID 是进程当前起作用的 ID，因此也称为现行用户 ID 和现行组 ID。当用户属于多个组时，附加组 ID 反映的是用户的其他组。系统使用这 3 个 ID 来确定进程是否能够访问某个文件。
- 保存的调整用户 ID 和保存的调整组 ID 含有程序被执行时的有效用户 ID 和有效组 ID 的副本。保存的调整用户 ID 是 POSIX.1 的选项。用户可以在编译时测试系统预定义宏_POSIX_SAVED_IDS 或在运行时以_SC_SAVED_IDS 参数调用 sysconf 来查看实现是否支持这一特征。

Linux 中进程之所以有这样三组 ID，就是为了让执行某个程序的进程得到适当的权限。例如 passwd 程序在执行时，它的实际用户 ID 将是执行该程序的某个普通用户的 ID，它的有效用户 ID 将是根用户的 ID。这样，普通用户便可以通过 passwd 程序来访问/etc/passwd 文件。保存的调整用户 ID 可以使得进程在有效用户 ID 和实际用户 ID 之间进行切换。

在大多数情况下，进程的有效用户 ID 就是实际用户 ID。

Linux 中所有文件都是由进程创建的，当文件首次被创建时，它的属主 ID 是创建它的

那个进程的有效用户 ID；它的组 ID 是该进程创建它时所在目录的组 ID。

在介绍完各种 ID 之后，我们看一下有关的系统调用函数 chown 和 fchown。

它们的格式如下：

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char * filename, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

其中，第 1 个参数 chown 是被设置的文件名(包含路径)，而 fchown 是文件描述符。第 2 个参数表示重置后的文件属主，第 3 个参数是重置后的文件所属组。

chown 和 fchown 函数用于改变文件的所有权关系，即可以改变文件的属主 ID 和所属组 ID。它们都改变指定文件的用户 ID 为 owner，组 ID 为 group，并且调用成功都返回 0，否则返回-1。

由于涉及到有关文件权限的问题，只有 root 用户才可以使用 chown 函数用来任意改变一个文件的所有者及其所属的组。而普通用户是没有这样的权限的。普通用户只能修改自己所有的文件的组识别号，且只能在其所属的组之中进行选择。

3.4.2.4 rename 函数

rename 函数用来对文件重命名。它的格式如下：

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

其中的参数 oldname 和 newname 都是字符串指针。分别代表旧文件名和新文件名。

调用成功时，返回值为 0；调用失败时，返回值为-1。

rename 调用是否能成功，将与 oldname 指向普通文件还是目录文件，newname 所表示文件是否已存在，若存在是普通文件还是目录文件等情况有关系，可以从表 3-6 中清楚地看出这些关系。

表 3-6 rename 的参数选项

	newname 所示文件不存在	newname 指向普通文件	newname 指向目录文件
oldname 指向普通文件	文件被重命名	newname 被删除，原来名为 oldname 的文件被重命名为 newname	错误
oldname 指向目录文件	文件被重命名	错误	newname 所指向的目录文件为空目录，则该目录文件被删除，oldname 被重命名，否则出错

对于修改目录文件的情况，有一点要注意。就是 newname 不能包含有 oldname 的路径前缀。也就是说，不能将一个目录文件重命名为它的子文件。

当 newname 和 oldname 指向同一个文件时，rename 调用不做任何操作而成功返回。

3.4.2.5 truncate 和 ftruncate 函数

有时需要对文件的大小进行修改。这时将会用到截断文件长度的函数 truncate 和 ftruncate。它们的格式如下：

```
#include <unistd.h>
int truncate(char *pathname, size_t len);
int ftruncate(int fd, size_t len);
```

其中的参数 len 用于指定要将文件截取到的长度。两个函数分别是针对文件路径和文件描述符使用的。

调用成功时，返回值为 0；调用失败时，返回值为-1。

3.4.2.6 access 函数

Linux 内核总是根据进程的有效用户 ID 和有效组 ID 来决定一个进程是否有权访问某个文件。当一个置有调整用户 ID 的程序运行时，它的有效用户 ID 可以与实际用户 ID 不同，特别是这种程序作为特权用户运行时，可以允许它访问超出用户范围的任何文件，例如，/etc/passwd。另一方面，这种程序也可能会需要访问由运行该程序的用户指定的文件。因此，在编写调整用户 ID 的程序时，在读写一个文件之前必须明确检查其用户是否原本就有对此文件的访问权限。为了实现这种确认，需要使用 access 函数。

access 函数的一般形式是：

```
#include <unistd.h>
int access(const char *pathname,int mode );
```

其中，pathname 是希望检验的文件名(包含路径)，mode 是欲检查的访问权限，如表 3-7 所示。

access 检查用户对一个文件的权限情况，根据 mode 的值检查调用进程对文件 pathname 是否具有读、写或执行的权限。若进程实际用户具有 mode 所指出的权限，access 返回 0，否则返回-1。

表 3-7 access 的访问权限选项

选　项	说　明
R_OK	检验调用进程是否有读访问权限
W_OK	检验调用进程是否有写访问权限
X_OK	检验调用进程是否有执行访问权限
F_OK	检验规定的文件是否存在

无论文件 pathname 是否允许进程对它读、写或执行，该调用都检查文件 pathname。如果文件 pathname 只是一个符号链接，则检查符号链接所指的文件。这个检查还依赖于文件所存在的目录，及参数 pathname 路径中包含的各个目录的存取权限。如果文件是符号链接，则依赖于符号链接所指文件所在的目录的存取权限。这个检查是按照进程的真实 UID 和 GID 来检查存取权限的，而不是按进程的有效用户 ID(EUID)检查的。这样就允许设置用户 ID(setuid)的程序可以容易地检查真正的用户授权。函数只检查存取权限，而不是文件类型或内容，即检查该进程对指定文件的读、写和执行的权力。比如，若检查目录有写权限，就意味着可以在这个目录里创建文件。

例如：

```
access("test",06);
access("test",F_OK);
```

分别用来检查实际用户对 test 文件是否具有读写权限和 test 文件是否存在。

3.4.3　查询文件信息

在应用程序中，经常要获取有关文件的状态信息，如一个文件的文件类型、大小、文件属主以及修改时间、或者一个文件的空闲块数、空闲节点数等。本小节将介绍获取单个文件状态信息的系统函数。

3.4.3.1　utime 和 utimes 函数

每一个文件都有 3 个时间戳与之相连：最近访问时间、最近修改时间以及最近特性修改时间，它们分别对应于 stat 结构的 st_atime、st_mtime 以及 st_ctime 成员。为简洁起见，在本书中简单地称它们为访问时间、修改时间和特性修改时间。

文件的访问时间 st_atime 给出最近一次读或者执行该文件的时间；修改时间 st_mtime 给出最近一次写该文件的时间，即文件内容被更改的时间；特性修改时间 st_ctime 给出文件的 inode 被改变的时间，如改变文件的访问权限、文件的属主、文件的链接数等。因为 inode 中的信息与文件的内容分别存储，因此除了文件的修改时间外，还需要特性改变时间记录关于 inode 的改变时间。

所有这些时间都用日历时间格式表示为 time_t 对象，time_t 数据类型定义在<time.h>中。

许多涉及文件的函数都会更新这 3 种时间，有时候它们仅更新其中一种或两种时间，有时候则不仅更新文件本身的时间，而且还更新也含文件的目录的时间。那么它们究竟更新的是其中的哪几种时间以及是谁的时间呢?其实这很好判断。只要我们能够分辨出函数对文件的操作涉及的是文件本身还是文件的 inode。一般而言，如果函数仅操作文件本身，则更新文件的访问或修改时间；如果函数操作涉及到文件的 inode 则更新文件的特性修改时间；如果函数操作涉及到了文件所在目录中的登记项，则还要更新这个目录的两个修改

时间。

例如，当创建一个新文件时，此文件的 3 个时间均设置为当前时间；此外，由于在包含该新文件的目录中要增加一个新的登记项，因而同时还要更新该目录的特性改变时间以及修改时间。又如，当调用 chmod 或 chown 改变文件的方式位或者文件的属主时，它仅涉及 inode 中的 st_mode 或 st_uid、st_gid，因此，只更新文件的特性修改时间 st_ctime。

可以利用 utime 函数来改变一个文件的访问时间和修改时间，但是没有函数可以改变文件的特性修改时间，因为 inode 是由系统来维护的。

它的格式如下：

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *pathname,const struct utimebuf *times );
```

其中，参数 pathname 指明要更新时间的文件(包含路径)，utimebuf 是专用于 utime 函数的结构类型，它含有表 3-8 所列成员，这两个成员之值均为日历时间。参数 times 为该文件指定新的访问时间和修改时间，它可以是空指针。

表 3-8　utimebuf 结构

成　员	说　明
time_t actime	文件的访问时间
time_t modtime	文件的修改时间

如果 times 是空指针，文件的访问时间和修改时间均设置为当前时间。此时，要么进程的有效用户 ID 必须等于文件的用户 ID，要么进程必须有该文件的写权限。

如果 times 不是空指针，它解释为指向 utimebuf 结构的指针并且用 times 值更新文件的访问时间和修改时间。在这种情形下，要么进程的有效用户 ID 必须等于文件的用户 ID，要么必须是超级进程。仅具有文件的写权限是不够的。

utime 调用成功将返回 0 并且自动更新文件的特性修改时间 st_ctime；否则返回-1。

Linux 另外还提供了一个与 utime 功能相同的函数 utimes，但它比 utime 具有更高的时间解析度，utimes 可以设置文件的访问和修改时间至微秒。

```
#include <sys/time.h>
int utime(const char *pathname,const struct timeval values[2] );
```

utimes 除了其第二参数与 times 不同之外，其余与 utime 完全相同。它的第二参数 values 是一个指向 timeval 结构的数组，此数组以微秒的精度指定文件 pathname 的访问时间和修改时间。其中，values[0]指定文件的访问时间，values[1]指定文件的修改时间。timeval 结构定义在头文件<sys/time.h>中，表 3-9 给出了其成员。

表 3-9　timeval 结构

成　　员	说　　明
long tv_sec	自公元纪年以来的秒数
long tv_usec	秒后的微秒数

例 3-3　打开一个文件，将它截断至 0 长度，但维持它的访问时间和修改时间不变。

```
1     #include <sys/types.h>
2     #include <utime.h>
3     #include <sys/stat.h>
4     #include <fcntl.h>
5     int main(int argc, char *argv[])
6     {
7         int i,fd;
8         struct stat statbuf;
9         struct utimbuf times;
10        if(argc!=2)
11        {
12          printf("Usage: a filename\n");
13          return 1;
14        }
15        if((fd=open(argv[1],O_RDWR))<0)             /*    打开文件   */
16        {
17          printf("%s open failed.\n",argv[1]);
18          return 3;
19        }
20    if(stat(argv[1],&statbuf)<0)
21          return 2;
22
23        if(ftruncate(fd,0)<0)                       /*    截断文件   */
24        {
25          printf("%s truncate failed.\n",argv[1]);
26          return 4;
27        }
28        printf("%s is truncated now.\n",argv[1]);
29        times.actime=statbuf.st_atime;              /*恢复文件时间至原时间*/
30        times.modtime=statbuf.st_mtime;
31        if(utime(argv[1], &times)==0)
32          printf("utime() call successful \n");
33        else
34          printf("Error:utime() call failed. \n");
35        return 0;
```

```
36    }
```

以上程序编译后的输出文件要命名为 a。程序实现的具体步骤为：打开文件(第 15~19 行)，然后用 stat 获取文件的当前时间并对它进行截断(第 20~27 行)，之后再调用 utime 恢复文件原来的时间(第 31~34 行)。

看看该程序的运行结果：

```
lxy@lxy-desktop:~$ ls -l text1                    /*查看文件大小和最近访问时间*/
-rw-r--r-- 1 lxy lxy 68 2007-12-31 15:53 text1
lxy@lxy-desktop:~$ ./a text1                      /* 运行程序 */
text1 is truncated now.
utime() call successful
lxy@lxy-desktop:~$ ls -l text1                    /* 检查结果 */
-rw-r--r-- 1 lxy lxy 0 2007-12-31 15:53 text1
```

从程序运行结果可以看出，文件 text1 的访问时间和修改时间都没有改变，但文件长度已经截断为 0。

3.4.3.2 stat、fstat 和 lstat 函数

utime 和 utimes 函数只能获取文件时间的相关信息，要获得文件的其他有关状态信息，需要调用 stat、fstat 和 lstat 函数。它们功能相似，但又有细微的不同。

它们的格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *pathname,struct stat *buf);
```

其中，参数 pathname 是指定的文件名(包含路径)，fd 是指定的文件描述符。buf 是指向 stat 结构的指针。

这些函数将返回指定文件的信息。调用这些函数的进程不需要任何对该指定文件的访问权限就可获得这些信息，但调用这些函数的进程需要对指定文件的路径有搜索的权限。stat 函数将文件 pathname 的信息存放在参数 buf 所指向的 stat 结构中。lstat 与 stat 功能相同，其唯一的区别是，对于符号链接文件，lstat 返回的是该符号链接本身的信息，而 stat 返回的是符号链接所指向文件的信息。而 fstat 也与 stat 的功能相同，区别仅在于 stat 使用文件名指向文件，而 fstat 用文件描述符指向文件。由于这个区别，fstat 可以获得非命名管道文件的状态信息，而 stat 则不能。有关非命名管道文件的说明见进程操作一章。这三个函数调用获得的信息都存放在一个 stat 结构中。

当函数调用成功时，返回值为 0，发生错误时则返回-1。

例 3-4 用 lstat 函数获取文件的类型。

```
1    #include <stdio.h>
2    #include <sys/types.h>
3    #include <sys/stat.h>
4    #include <fcntl.h>
5
6    int main(int argc, char *argv[])
7    {
8        int i;
9        struct stat buf;
10       char    *ptr;
11       for(i=1; i<argc;i++)
12       {
13         printf("%s: ",argv[i]);
14         if(lstat(argv[i],&buf)<0)
15         {
16               printf("error! \n");
17               continue;
18         }
19         switch(S_IFMT&buf.st_mode)          /* 测试文件类型 */
20         {
21               case S_IFREG: ptr="regular"; break;                  /* 普通文件 */
22               case S_IFDIR:   ptr="directory"; break;            /* 目录文件 */
23               case S_IFCHR: ptr="character special"; break;      /* 字符设备文件 */
24               case S_IFBLK: ptr="block special"; break;            /* 块设备文件 */
25               case S_IFFIFO: ptr="fifo"; break;                   /* 管道文件 */
26               case S_IFSLNK: ptr="symbolic   link"; break;         /* 符号链接文件 */
27               case S_IFSOCK: ptr="socket"; break;                 /* socket 文件 */
28               default:ptr="unknown mode ";
29         }
30         printf("%s \n",ptr);
31       }
32       return 0;
33   }
```

该程序从命令行读入输入参数，然后针对每一个命令行参数打印其文件类型。程序运行结果如下：

```
lxy@lxy-desktop:~$ ls –l      /* 列出文件类型
-rw-r--r-- 1 lxy   lxy         48 2008-01-01 15:21 text
drwxr-xr-x 4 lxy   lxy       4096 2008-01-26 22:47 test
lrwxrwxrwx 1 lxy   lxy         26 2007-12-13 03:57 Examples -> /usr/share/example-content
```

以上 3 个文件分别是普通文件，目录文件和符号链接文件，运行以上程序后，显示如下：

```
lxy@lxy-desktop:~$ ./lstat text test Examples
text: regular
test: directory
Examples: symbolic link
```

3.4.4 文件其他操作

文件是操作系统中一个十分重要的概念，因此，关于文件的操作是多种多样的。在接下来的部分中，将介绍其他一些常用的文件操作。

3.4.4.1 dup 和 dup2 函数

由一次 open 打开的文件可以有多个描述符与它相连，这些与同一次打开的文件相连的描述符称为重复描述符。我们可以用函数 dup 或 dup2 复制文件描述符。

它们的格式如下：

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

其中，oldfd 和 newfd 分别为复制前的文件描述符和复制后的文件描述符。

这两个函数调用都将复制文件描述符 oldfd。也就是说，新得到的文件描述符和原来的文件描述符将共同指向一个打开的文件。两个调用的返回值都为新的文件描述符，不同的是，系统调用 dup 的返回值是最小的未用文件描述符，而系统调用 dup2 的返回值是预先指定的文件描述符 newfd。如果文件描述符 newfd 正在被使用，则先关闭 newfd。如果 newfd 同 oldfd，则不关闭该文件正常返回。

通常使用这 2 个函数来重定向一个已打开的文件描述符。

例 3-5 使用 dup2 对文件描述符进行重定向。

```
1       #include <stdio.h>
2       #include <unistd.h>
3       #include <fcntl.h>
4       #include <sys/stat.h>
5       #include <sys/types.h>
6
7       int main(int argc, char * argv[])
8       {
9               int fd;
10              if(argc<2)
11              {
```

```
12              printf("useage dup2    filename.\n");
13              return 1;
14          }
15          if((fd=open(argv[1],O_WRONLY|O_CREAT,0644))==-1)
16          {
17              printf("Open file %s errof!\n",argv[1]);
18              return 2;
19          }
20          if(dup2(fd, STDOUT_FILENO)==-1)
21          {
22              printf("dup2 fd failed!\n");
23              return 3;
24          }
25          printf("dup2()    call successed!\n");
26          close(fd);
27          return 0;
28      }
```

在上面的程序中首先用 open 函数打开命令行输入的文件，然后用 dup2 函数将标准输出文件重新定向到打开的文件中。运行这个程序可以看到 printf 输出的字符串不在终端而在命令行输入的文件中。下面是程序运行结果：

```
lxy@lxy-desktop:~/test1$ ls -l text            /*查看文件 text 文件大小  */
-rw-r--r-- 1 lxy lxy 1 2008-02-26 21:39 text
lxy@lxy-desktop:~/test1$ ./dup2 text             /*    运行程序        */
lxy@lxy-desktop:~/test1$ ls -l text            /*再次查看 text 文件  */
-rw-r--r-- 1 lxy lxy 24 2008-02-26 21:40 text
lxy@lxy-desktop:~/test1$ cat text              /*  查看 text 文件内容  */
dup2()    call successed!
```

3.4.4.2 fcntl 函数

函数名 fcntl 代表文件控制(file control)，它提供了进一步管理低级文件描述符的各种手段，用它可以对已打开的文件描述符执行各种控制操作。例如，重复一个文件描述符，查询或设置文件描述符标签，查询或设置文件状态标签，操纵文件锁等。

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, int arg)
```

fcntl 函数是对已打开文件的文件描述符进行各种控制操作。根据参数 cmd 的值决定是否要第三个附加参数 arg。

参数 cmd 可以指定的选项如表 3-10 所示。

表 3-10　cmd 的选项

选　项	说　明
F_DUPD	返回大于或等于 arg 的最低序号的文件描述符。该功能可以由 dup 函数实现。新的文件描述符与旧的可以互换使用。调用成功，返回值为新的文件描述符
F_GETFD	获得 close-on-exec 标志。如果最后一位是 0，则该标志没有设置。返回值为 0 或 1
F_SETFD	设置 close-on-exec 标志为指定的值 arg(只有最后一位有效，为 0 或 1)
F_GETFL	获得文件打开的方式。返回所有的标志位，标志位的含义与 open 一节相同
F_SEGFL	设置文件打开方式的标志。设置文件打开方式为参数 arg 指定的方式。仅能设置 O_APPEND 和 O_NONBLOCK(或 O_NDELAY)，有的系统还可以设置 O_SYNC。该标志被文件描述符所有的副本(dup 函数或 fcntl 的 F_DUPFD 产生)所共享
F_GETLK	获得本进程得到锁的第一个锁的 flock 结构
F_SETLK	获得离散的文件锁，不等待
F_SETLKW	获得离散的文件锁，必要时等待
F_GETOWN	返回当前接收 SIGIO 或 SIGURG 信号(signal)的进程 ID 或进程组。进程 ID 以负值返回
F_SETOWN	设置进程或进程组接收 SIGIO 和 SIGURG 信号，进程组 ID 以负值返回。进程 ID 用正值指定

flock 结构定义如下：

```
struct flock{
        long  l_start;          /*  块开始处的偏移量 starting offset */
        long        l_len;      /*  块长  */
        long        l_pid;      /*  锁的属主(进程)*/
        long        l_type;         /*  锁的类型：读/写等*/
        long        l_whence;       /*  块开始处的类型  */
};
```

关于加锁和解锁区域的说明还要注意以下各点：

(1) 该区域可以在当前文件尾端处开始或越过其尾端处开始，但是不能在文件起始位置之前开始或越过该起始位置。

(2) 如若 l_len 为 0，则表示锁的区域从其起点(由 l_start 和 l_whence 决定)开始直到最大可能位置为止。也就是不管填写到该文件中多少数据，它都处于锁的范围。

(3) 为了锁整个文件，通常的方法是将 l_start 说明为 0，l_whence 说明为 0，l_len 说明为 0。

在修改文件描述符标志或文件状态标志时必须谨慎，先要取得现在的标志值，然后按

照希望修改它，最后设置新标志值。不能只是执行 F_SETD 或 F_SETFL 命令，这样会关闭以前设置的标志位。

fcntl 函数在执行失败时返回值为-1。

例 3-6 下面的程序使用 fcntl 改变打开的一个文件的属性。

```
1	#include <sys/types.h>
2	#include <sys/stat.h>
3	#include <fcntl.h>
4	#include <stdio.h>
5	#include <unistd.h>
6	#define DUMMY 0
7	#define BUFSIZE 1024
8
9	char *validID[]={"123\n", "5678\n", "007\n", "421\n", "F"};
10
11	char str1[]={"1234"}, str2[]={"1234"};
12
13	int main(int argc, char *argv[])
14	{
15		int fd, flags;
16		int n;
17		char userno[BUFSIZE], **ptr;
18		ptr=validID;
19		printf("%d \n",strcmp(str1,str2));
20		setbuf(stdout,(char *) NULL);
21		if((fd=open("/dev/tty", O_RDONLY | O_NDELAY))==-1)
22		{
23			printf("open error!\n");
24			return 1;
25		}
26		printf("Enter your user ID :\n");
27		sleep(3);
28		if(read(fd, userno,BUFSIZE)==0)
29		{
30			printf("Bye Bye!\n");
31			return 2;
32		}
33		while((strcmp(*ptr, userno)!=0)&&(strcmp(*ptr,"F")!=0))
34			++ptr;
35		if(strcmp(*ptr,"F")==0)
36		{
37			puts("Invalid user ID\n");
38			return 3;
```

```
39            }
40            flags=fcntl(fd, F_GETFL,DUMMY);
41            flags&=fcntl(fd, F_SETFL,flags);
42            printf("Enter your department Number:\n");
43            n=read(fd, userno, BUFSIZE);
44            printf("\n Welcome to Department #");
45            write(1, userno, n);
46            close(fd);
47            return 0;
48    }
```

这个程序用 open 函数打开一个终端，从这里输入数据。由于在打开时，O_NDELAY 标志被设定，因此当随后的 read 函数读不到数据时会立即返回，而不会被挂在那里一直等待。在 open 函数之后程序会暂停 3 秒钟再开始执行 read 函数，在这段时间里只要用户有输入，那么程序就会继续，否则输入程序就会中止。接下来程序检查输入的数据是否正确，如果数据有问题。程序会在显示错误信息后中止。程序用 fcntl 函数把 O_NDELAY 标志取消。这样一来 read 函数在用户未输入部门代码前会一直等待下去，这种做法与前面的 read 函数的执行情况完全不同，仅需对这两种操作方式稍作比较，便知 O_NDELAY 标志的作用了。

3.4.5　目录文件操作

Linux 系统中经常遇到的一个问题是扫描目录，例如，确定某个文件是否位于一个特定的目录中。Linux 提供了一组用于目录操作的标准函数，这些函数使得扫描目录的工作较为容易且有较好的可移植性。这一节介绍这些函数。

3.4.5.1　getwd 函数

每一个进程都有一个目录与之相连，它称为当前工作目录，或简单地称为工作目录。这个目录就是文件相对路径名的搜寻开始点。当用户注册到 Linux 系统并开始一个会话期时，当前工作目录初始设置成与用户注册账号相连中的主目录，通常即/etc/passwd 文件中用户登记项给出的主目录。当前工作目录是进程的属性，主目录是注册用户名的属性。用 shell 命令 cd 可以改变当前工作目录。

在程序中，用 getwd 函数可以获得进程的当前工作目录。

它们的格式如下：

```
#include <unistd.h>
char *getwd(char *pathbuf);
char *gectwd(char *pathbuf, size_t size);
```

getwd 函数确定调用进程当前工作目录的绝对路径名，复制该路径名于 pathbuf 所指、由我们自己提供的字符数组中，然后返回指向该数组的指针。所提供的这个字符数组长度至

少应当大于 PATH_MAX，这是定义在<limits.h>中系统规定的最大路径名长度。如果当前工作目录的路径名长度大于 PATH_MAX+1(包括 null 字符)，getwd 将失败并返回一个空指针。

getcwd 函数的作用与 getwd 相同，不同的是它给出了另一个参数 size 指明存放路径名字符数组的大小。

3.4.5.2 chdir 和 fchdir 函数

chdir 和 fchdir 函数用于重新指定调用进程的当前工作目录。

它们的格式如下：

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);
```

其中，pathname 是新的工作目录，函数 chdir 使得由 pathname 指定的目录成为调用进程的当前工作目录，调用成功，返回值为 0；调用失败则返回-1。调用 chdir 函数的进程必须有 pathname 所有路径分量的执行权限，并且 pathname 指定的路径长度不能超过 PATH_MAX，其路径分量不能超过 NAME_MAX。

fchdir 函数与 chdir 作用相同，不同的只是新的工作目录由打开的文件描述符 fd 指定。

例 3-7 改变进程的当前目录到指定的目录然后打印出该目录名。

```
1     #include <unistd.h>
2     #include <limits.h>
3
4     int main(int argc, char *argv[])
5     {
6     char path[1000];
7     if(argc!=2)
8     {
9             printf("Usage chdir <pathname>\n");
10            return 1;
11    }
12    if(chdir(argv[1])<0)
13    {
14            printf("chdir failed\n");
15            return 2;
16    }
17    if(getwd(path)==NULL)
18    {
19            printf("getwd failed\n");
20            return 3;
21    }
22    printf("CWD =%s \n", path);
```

```
23    return 0;
24    }
```

程序的运行结果为：

```
lxy@lxy-desktop:~/test1$ pwd              /* 查看 shell 当前工作目录 */
/home/lxy/test1
lxy@lxy-desktop:~/test1$ ./chdir ../test  /* 改变进程的当前工作目录 */
CWD =/home/lxy/test
```

应当注意的是：chdir 改变的是调用进程的工作目录，它对父进程的工作目录没有影响。因此程序终止运行回到 shell 时，shell 的当前工作目录仍为原来的工作目录。

3.4.5.3 mkdir 和 rmdir 函数

mkdir 函数用于创建目录。它的格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
```

其中参数 pathname 是新创建目录的目录名，mode 指定该目录的访问权限，这些位将受到文件创建方式屏蔽(umask)的修正。

该函数创建一个名为 pathname 的空目录，此目录自动含有“.”和“..”2 个登记项。这个新创建目录的用户 ID 被设置为调用进程的有效用户 ID，其组 ID 则为父目录的组 ID 或者进程的有效组 ID。

若调用成功，mkdir 将更新该目录的 st_atime、st_ctime 和 st_mtime，同时更新其父目录的 st_ctime 和 st_mtime，然后返回 0。若调用失败，mkdir 将返回-1。

由 pathname 指定的新目录的父目录必须存在，并且调用进程必须具有该父目录的写权限以及 pathname 涉及的各个分路径目录的搜寻权限。

rmdir 函数用于删除一个空目录，它的格式如下：

```
#include <unistd.h>
int rmdir(const char *pathname);
```

使用 rmdir 函数时，目录必须为空，否则调用失败，函数返回-1。成功时，函数返回 0。

例 3-8 创建一个新目录，然后删除此目录。

```
1    #include <sys/types.h>
2    #include <unistd.h>
3    #include <limits.h>
4    #include <sys/stat.h>
5
```

```
6    int main(int argc, char *argv[])
7    {
8    char path[1000];
9    char file[1000];
10   if(argc!=2)
11   {
12           printf("Usage mk <pathname>\n");
13           return 1;
14   }
15   getwd(path);                              /*取得当前工作目录 */
16   printf("current directory is :%s \n",path);
17   if(mkdir(argv[1],S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH)<0) /*创建新目录 */
18   {
19           printf("mkdir failed \n");
20           return 2;
21   }
22   if(chdir(argv[1])<0)                      /*改变当前工作目录为新目录 */
23   {
24           printf("chdir failed \n");
25           return 3;
26   }
27   getwd(path);
28   printf("mkdir successed.\n New current directory is: %s\n",path);
29   /*rmdir(path);                                  /*删除新建目录 */
30   printf("%s is removed\n",path); */
31   return 0;
32   }
```

以上程序首先用 getwd 函数取得当前工作目录，然后在当前工作目录下，利用 mkdir 函数创建新目录。新目录创建成功后，改变当前工作目录为新目录。程序执行结果如下：

```
lxy@lxy-desktop:~/test1/$ ls-l              /*查看当前目录内容 */
-rwxr-xr-x 1 lxy lxy 7291 2008-02-27 05:14 mk
lxy@lxy-desktop:~/test1/$ ./mk temp             /*执行程序，创建 temp 目录 */
current directory is :/home/lxy/test1
mkdir successed.
 New current directory is: /home/lxy/test1/temp
lxy@lxy-desktop:~/test1/temp$ ls-l          /* 再次查看当前目录内容 */
-rwxr-xr-x 1 lxy lxy 7291 2008-02-27 05:14 mk
drwxr-xr-- 2 lxy lxy 4096 2008-02-27 05:17 temp
```

从结果可以看出，程序执行后，当前目录下多了一个 temp 子目录。

把上述程序中的注释部分去掉后，再重新编译、执行，可以把新创建的目录删除掉。

3.4.5.4 opendir 函数

当访问一个目录文件时，同普通文件一样，需要将其打开。打开目录文件的函数是opendir。它的格式如下：

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *pathname);
```

其中，参数 pathname 是要打开目录的路径名。函数返回值是 DIR 类型，是指向目录文件的结构指针。调用成功时，返回值为一个目录指针；调用失败时，返回值为 NULL。

3.4.5.5 closedir 函数

关闭一个已打开目录文件的函数为 closedir。它的格式如下：

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dp);
```

其中，参数 dp 是要关闭的目录文件的指针，它是由 opendir 函数调用时获得的。

函数调用成功时，返回值为 0，调用失败时，返回值为-1。

3.4.5.6 readdir 函数

Linux 系统提供了读取一个目录文件内容的函数 readdir。它的格式如下：

```
#include <sys/types.h>
#include <dirent.h>
struct direct *readdir(DIR *dp);
```

其中，参数 dp 是指向要访问目录文件的指针。

函数调用成功，返回值为指向 dirent 的结构指针。dirent 结构的定义如下：

```
struct dirent
{
 ino_t d_ino;
 char d_name[NAME_MAX+1];
}
```

其中，d_ino 用于表示该目录的节点号。d_name 用于存放此目录链接的文件名。NAME_MAX 为系统预定义的常数。

函数调用失败，返回值为 0。

例 3-9 下面的程序说明了目录打开、读取和关闭函数的使用方法。

```
1	#include <sys/types.h>
```

```
2   #include <dirent.h>
3   #include <unistd.h>
4
5   int main(int argc, char *argv[])
6   {
7       char path[1000];
8       DIR * dp;
9       struct dirent *pdirent;
10      if(argc!=2)
11      {
12          printf("Usage   a <pathname>\n");
13          return 1;
14      }
15      if((dp=opendir(argv[1]))==NULL)
16      {
17          printf("Opendir %s failed\n", argv[1]);
18          return 2;
19      }
20      if((pdirent=readdir(dp))==0)
21      {
22          printf("readdir %s failed\n", argv[1]);
23          return 3;
24      }
25      printf("%s\n",pdirent->d_name);
26  closedir(dp);
27      return 0;
28  }
```

程序首先用 opendir 函数打开指定的目录，然后用 readdir 函数读取目录内容，并打印出所读目录内容，最后用 closedir 函数将刚才打开的目录文件关闭。

3.4.6 特殊文件操作

在 Linux 系统中，除去普通文件、目录文件外，还有其他比较特殊的文件。在这一节里，介绍其他文件的相关操作。

3.4.6.1 mknod 函数

利用 creat 和 open 函数只能建立一般文件，要建立其他类型的文件就要用 mknod 函数了。它的格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>
int mknod (const char *pathname, mode_t mode, dev_t dev);
```

其中，mode 指定创建的特殊类型文件的用户权限和文件类型。文件的权限可以通过一般的方法用进程的 umask 来修改，创建的特殊类型文件的默认权限是(mode&~umasks)。文件类型可以是 S_IFREG、S_IFCHR、S_IFBLK、S_IFIFO。如果文件类型是 S_IFCHR 或 S_IFBLK，那么还要指定设备的主设备号和次设备号，如果没有指定将被忽略。

新建的特殊类型文件将具有进程的有效 UID。如果包含这个特殊类型文件的目录设置了组 ID 位，那么这个文件将从上一级目录继承组的所有者权限。

参数 dev 只有在建立设备文件时才会用到，高字节指定主设备，低字节指定次设备。假设要建一个设备文件，其主设备号为 03，次设备号为 05，那么函数调用如下：

```
mknod("dev01",S_IFCHR | 0644, (03<< 8)| 05);
```

对于一般用户来说只能用 mknod 函数建立命名管道，如果要建立目录或设备文件则需要有超级用户的权力才行。

如果 mknod 函数执行成功，则返回 0，否则返回-1。

3.4.6.2 mount 与 umount 函数

通过前面对 Linux 文件系统的介绍，我们已经了解到在 Linux 中整个文件系统不一定在同一个硬盘上。它可以用挂接的方式将一个设备挂在另一个设备之下，这个设备可以是磁带、磁盘或软盘。一个文件系统挂接到另一个文件系统上后，它就如同一个一般的目录，对于用户来说这个目录与其他的目录没有什么区别，唯一限制是用户不能对位于不同设备的文件进行符号连接。mount 函数就是完成这个挂接的任务，它的格式如下：

```
#include <sys/mount.h>
#include <linus/fs.h>
int mount(const char *specialfile, const char *dir, const char *filesystemtype, unsigned long
            rwflag, const void *data);
int umount(const char *specialfile);
int umount(const char *dir);
```

mount 函数将由 specialfile 指定的文件系统挂接到由 dir 指定的目录下。umount 函数将由 specialfile 指定的文件系统从系统中卸载。只有超级用户才能使用 mount 和 umout 来挂接和卸载文件系统。

参数 filesystemtype 可能使用在文件/proc/filesystems 中列出的文件系统。系统中/proc/filesystems 的内容可能如下：

```
    ext3
nodev proc
        iso9660
```

```
nodev devpts
```

这几个文件系统在/etc/fstab 中定义。另外还可以使用其他类型的文件系统。参数 rwflag 在高 16 位是 magic number 0xC0ED，另外低 16 位的各种挂接时的标志在/linux/fs.h 中定义。如果这个参数的 magic number 没有指定，那么 mount 的后面两个参数是没有用的。参数 data 由不同的文件系统来使用。

这个函数如果调用成功，返回 0，否则返回-1。

3.4.6.3 链接与 link 函数

Linux 文件系统提供了一种将不同文件名链接至同一个文件的机制，这种机制称为链接，它可以使得单个程序对同一文件使用不同的名字。这样做的好处是文件系统只存储文件的一个副本。

系统简单地通过在目录中建立一个新的登记项来实现这种链接，该登记项具有一个新的文件名和要链接文件的 inode 号。文件的目录登记项也称为文件的硬链接，简称为链接。任何文件可以有一至多个链接与之相连，这些链接可以在同一目录，也可以在不同目录，它们之间没有主次之分。不论一个文件有多少链接，在磁盘上只有一个描述它的 inode。只要文件的链接数不为 0，该文件就保持存在。每个文件的链接数记录在 inode 的 st_nlink 域。文件所允许的最大链接数由 LINK_MAX 定义。

给文件建立一个链接(也即给文件增加一个新名字)，使用 link 函数，它的格式如下：

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

其中，oldpath 是已存在的文件名，newpath 是为原有文件建立的链接名。

link 用 newpath 给出的名字创建一个新的目录登记项，该登记项引用现存文件 oldpath。它成功时返回 0，否则返回-1。

例 3-10 下面的程序说明了 link 函数的作用以及如何用它建立与现存某个文件的链接。

```
1     #include <stdio.h>
2     #include <stdlib.h>
3     #include <sys/stat.h>
4     int main(int argc, char *argv[])
5     {
6             int link_value, fd;
7             const char *path="linkfile";
8             struct stat orig_buf, new_buf;
9             if(argc!=2)
10            {
11                    printf("Usage a <pathname>\n");
12                    return 1;
13            }
```

```
14          printf("create newfile\n");
15          if((fd=creat(argv[1]), S_IRWXU | S_IRWXG | S_IRWXO)==-1)
16          {
17                  printf("create file   %s failed\n", argv[1]);
18                  return 2;
19          }
20          printf("Get newfile status \n");
21          stat(argv[1], &orig_buf);
22          printf("orig_buf.st_nlink=%d \n", orig_buf.st_nlink);
23          printf("create link from %s to %s \n", argv[1],path);
24          if(link(argv[1],path));
25          {
26                  printf("link call failed\n");
27                  return 3;
28          }
29          printf("link call successful \n");
30          stat(argv[1],&new_buf);
31          printf("new_buf.st_nlink=%d\n",new_buf.st_nlink);
32          return 0;
33  }
```

该程序首先用 creat 函数创建一个新文件，用 stat 函数取出文件的状态信息，并打印输出状态信息中的文件链接数。然后用 link 函数为新文件建立一个名为 linkfile 的链接，并再次输出文件的链接数。

该程序的执行结果如下所示。在链接之前文件 newfile 的链接数为 1，当调用 link 链接了 linkfile 之后，其链接数变为 2。

```
lxy@lxy-desktop:~/test1$ ./a newfile              /* 执行程序 */
create newfile
Get newfile status
orig_buf.st_nlink=1
create link from newfile to linkfile
link call successful
new_buf.st_nlink=2
```

3.4.6.4 符号链接与 symlink 和 readlink 函数

符号链接也称为软链接，它是指向另一个文件的特殊文件，这种文件的数据部分仅包含它所要链接文件的路径名。路径名可以是绝对路径也可以是相对路径，如果是相对路径名，它解释为相对于包含此符号链接的目录。

符号链接与硬链接不同，它不直接使用 inode 号作为文件指针，而是使用文件的路径名作为指针。符号链接有自己的 inode，并且在磁盘上有一小片空间存放路径名。正因为符

号链接用路径名而不是用 inode 号作为链指针，因此它比硬链接更为灵活，它可以跨文件系统，也可以与目录链接。

符号链接与硬链接不同之处还有：符号链接可以对一个不存在的文件名进行链接，不过，直到这个名字对应的文件被创建之后，才能打开其链接。类似地，如果符号链接指向一个已存在的文件，而后该文件被删除，则符号链接仍然指向相同的文件名，尽管此名字已不再代表任何文件。

可以用函数 symlink 创建符号链接。它的格式如下：

```
#include <unistd.h>
int symlink(const char *path1, const char *sympath);
```

其中，参数 path1 是原始文件名，sympath 是符号链接文件名。

symlink 创建一符号链接文件 sympath，该文件指向 path1。此时并不要求 path1 是已存在的文件，并且也不要求 path1 和 sympath 属于同一文件系统。

symlink 的正常返回值为 0，出错时为-1。

符号链接虽然像普通文件或目录一样有名字也有位置，但它不像普通文件或目录，它没有实在的文件内容；它的文件内容只是指向另一个文件或目录的指针。当系统打开一个一般文件时，它便可直接读取文件的内容。然而系统打开符号链接时，仅从链中读取路径，然后顺链接路径直至找到实际文件，再读取所指向的文件或目录。

为了打开符号链接文件本身，系统操供了 readlink 函数。

```
#include <unistd.h>
int readlink(const char *pathname, char *buf, int bufsize);
```

其中，pathname 是符号链接文件名，buf 是字符缓冲区，参数 bufsize 给出要复制的最大字符个数，通常情况下它就是 buf 所指存储空间的大小。

readlink 将打开文件、读文件以及关闭文件 3 个动作集成在一起。如果调用成功，它存储 pathname 给出的符号链接文件的内容(即此符号链接所指的文件名)于 buf 中。注意，此文件名字符串不是空字符终止的。

readlink 的正常返回值是 buf 中实际存放的字符个数，其失败时的返回值为-1。

3.5　小　　结

本章介绍了有关文件和文件系统的概念。学习了文件的基本输入输出操作，还介绍了其他与文件操作相关的函数，包括确定和改变文件模式、查询文件信息、目录文件的操作等。同时还介绍了 Linux 平台下的特殊文件的有关操作。

熟悉 Linux 平台下的文件和文件系统以及相关的函数，对更好地在 Linux 平台下编写

C 程序是非常必要的，希望读者认真掌握。

习　　题

一、填空题

1. Linux 的文件是个简单的______。

2. 对于 Linux 而言，所有对设备和文件的操作都使用______来进行。

3. 调用______函数可以打开或创建一个文件。

4. 设置文件的存取权限，分为______、______和______三类。每类分为______、______和______权限。

5. 每一个进程都有一个目录与之相连，它称为______，或简单地称为______。

二、选择题

1. ______函数是从文件中读取指定长度的数据到内存中。

(A) open　(B) read　(C) write　(D) create

2. ______函数是将内存中的数据写入文件。

(A) open　(B) read　(C)　write　(D) create

3. 假设用户 user 是 file1 文件的拥有者，file1 文件的存取权限被设为-r-xr—r--，这表明 user 只有______的权力。

(A) 读和执行　(B) 读和写　(C) 写和执行　(D) 只读

4. ______函数提供了进一步管理低级文件描述符的各种手段，用它可以对已打开的文件描述符执行各种控制操作。

(A) chmod　(B) fcntl　(C) chown　(D) umask

5. 当访问一个目录文件时，同普通文件一样，需要将其打开。打开目录文件的函数是______。

(A) closedir　(B) mkdir　(C) opendir　(D) readdir

三、上机题

1. 编写一个程序，打开一个文本文件，读取其中内容，将其复制到一个新建文件中。

2. 编写一个程序，打开一个文件，将它截断至原来长度的 1/2。

3. 编写一个程序，打开一个文本文件，然后把此文件中小写字母转换为大写字母，其他字符不变。其中文件名作为命令行参数。

4. 编写一个程序，读取当前工作目录下的内容，并将其打印输出到终端。

CHAPTER 4

标准 I/O 库

前面的一章介绍的文件输入输出操作方法都是基于文件描述符的。在这一章里，将介绍另一种输入输出方法——基于流的标准输入输出操作。

与基于文件描述符的输入输出相比，基于流的标准输入输出更加简单、方便，因而在C程序的编写中，被广泛地使用。通过本章的学习，读者将对基于流的标准输入输出操作有所了解。

4.1 概　　述

标准输入输出的函数都存在于标准输入输出库中。这些库函数处理了输入输出的一些细节，比如缓冲区的分配、设置最佳的缓冲区大小等，为用户避免了考虑这些琐事的烦恼，使用起来更简单，更方便。

另外，标准输入输出库是使用标准C语言书写的，所以它不光可以应用于Linux系统中，也可以用于其他的系统(如Windows系统)。而基于文件系统的输入输出与操作系统有很大关系，不能应用于其他系统。因此，标准输入输出的应用范围更广泛。

4.2 流和 FILE 对象

上一章中介绍的输入输出系统函数都是围绕着文件描述符展开的。当打开一个文件时，这个文件描述符也被返回，并被应用于接下来的输入输出操作。而标准输入输出却是

围绕“流”(stream)进行的。当调用标准输入输出打开或创建一个文件时，就将一个“流”与这个文件关联在一起了。一旦流被打开，就可以读写它了。

在对流进行操作之前，需要将它打开。当打开一个流时，标准输入输出函数返回一个FILE结构的指针。这个结构中包含所有的对这个流进行操作所需要的信息。如：真实输入输出操作所需的文件描述符，为这个流而准备的缓冲区的指针，缓冲区的大小，当前缓冲区的字符数，出错标志等等。应用程序并不需对 FILE 结构的内容有任何的了解，只需要将一个特定的 FILE 结构指针传递给标准输入输出函数就可以了。通过这个结构，系统函数自动的对特定的流进行操作，而不需用户的任何干预。

有三个流是在执行程序时自动打开的。它们是标准输入、标准输出和标准错误输出。相应的 FILE 结构指针为 stdin、stdout、sderr，它们和 STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO 文件描述符对应的文件是相同的。当对流完成操作后，需要通过操作系统清空缓冲区、保存数据等，可以通过调用系统函数 close 来实现这些操作。如果不关闭流，就有可能造成数据的丢失。标准输入、标准输出、标准错误输出是自动关闭的。

4.3　打开和关闭流

用于打开流的函数是 fopen、freopen、fdopen。fopen 打开一个文件便创建了一个新的流与该文件的连接，如果该文件不存在，则创建一个新文件。freopen 在一个特定的流上打开一个特定的文件。fdopen 函数可以把一个文件指针和文件描述符 fd 关联。这几个系统函数的格式如下：

```
#include <stdio.h>
FILE* fopen(const char *pathname, const char * mode);
FILE* freopen(const char *pathname, const char *mode, FILE *fp);
FILE* fdopen(int fd, const char *mode);
```

其中，参数 pathname 是包含文件路径的文件名，mode 参数指定对流文件的读写方式，fp 是流结构指针。fd 是文件描述符。

函数 fopen 打开由 pathname 指定的文件并创建一个与之相连的流。

函数 freopen 在一个特定的流上打开一个特定的文件。首先关闭由 FILE *fp 指定的已打开的流，然后打开由 pathname 指定的流。它一般用于打开一个特定的文件以代替标准输入流、标准输出流、标准错误输出流等预定义流。

函数 fdopen 将一个流对应到某个已打开的文件上，fd 就是这个文件的描述符。这个调用只有在已打开文件的模式和 mode 定义的模式相同的情况下才能成功。这个已打开的文件一般是管道文件或网络通信管道，这些文件没有办法通过标准输入输出函数 fopen 打开，只有道过调用特定设备函数得到文件描述符，然后通过 fdopen 函数将一个流与这个文件关

联起来。

如果打开文件成功，它们返回指向此流的指针，否则返回空指针 NULL。

mode 参数是一字符串，称为打开方式参数，它控制流打开的方式。表 4-1 列出了常用流打开方式选项。

表 4-1　流打开方式

选　　项	说　　明
r 或 rb	为只读而打开一个已存在的文件
w 或 wb	为只写而打开文件。如果该文件存在，则将它的长度截为 0，也即该文件将被重新写过；如果该文件不存在，则创建一个新文件
a 或 ab	为在文件末尾添加内容而打开文件。若文件已经存在，其原来的内容不变且到该流的输出将添加在文件的末尾；否则，创建一个新的空文件
r+或 rb+或 r+b	为更新(即，既读又写)而打开一个已存在的文件，该文件的原内容不变且初始文件位置位于文件开始之处
w+或 wb+或 w+b	为更新而打开一个文件。若文件已存在，其长度被截至 0；否则，创建一个新文件
a+或 ab+或 a+b	为更新而打开一个文件。若文件已存在，其原内容不变；否则，创建一个新文件。用于读的初始文件位置位于文件开始之处，但输出总是添加到文件的末尾

其中，字母 r 代表 read，字母 w 代表 write，字母 a 代表 append。字母 b 代表 binary，它指明要求打开的是一个二进制文件而不是文本文件。在区分二进制文件和文本文件的操作系统中，字母 b 有这里指定的意义。但是 Linux 系统并不区别所打开流的内容是二进制文件还是正文文件，因此字母 b 实际上没有作用。

字符“+”指明为更新而打开一个文件。当以这种方式打开一个文件时，对它既可写也可读。但是在从读转变为写或者从写转变为读时，不能紧接在读之后进行写操作或者反之。在它们之间必须调用 fluab 函数或者文件定位函数(fseek，fsetpos 或 rewind)，否则会由于内部缓冲区可能未被清空而出现意料之外的错误。

对于以添加方式(即打开方式参数是“a”或“a+”)打开的文件，不可能覆盖该文件原来的内容，可以用 fseek 函数定位文件指针于文件的任何位置来读取该文件。但是当写该文件时，当前文件指针被忽略，所有写至该文件的内容都添加在文件尾并且文件指针被重定位至新写入内容的末尾。

打开的流通过调用 fclose 函数来关闭。它的格式如下：

```
#include <stdio.h>
int fclose(FILE *fp);
```

其中，参数 fp 是流指针。fclose 使得参数 fp 指定的流被关闭并且中断它与对应文件的连接。在流被关闭之前，所有缓冲的输出将被写出，所有缓冲的输入将被丢弃。如果标准

I/O 库为流自动分配了缓冲区，该缓冲区将被释放。

fclose 调用成功返回 0，否则返回 EOF。

此外，fcloseall 函数可以关闭除 stdin 或 steout 外的流。它的格式如下：

```
#include stdio.h
int fcloseall();
```

函数成功则返回关闭文件流的数量，失败则返回 EOF。

例 4-1 fopen 和 fclose 函数的使用方法。

```
1    /* ex1.c */
2    #include <stdio.h>
3    int main(int argc, char *argv[])
4    {
5    FILE *fp;
6    int          iflag;
7    if(argc<=1)
8    {
9         printf("usage: %s filename\n",argv[0]);
10        return 1;
11   }
12   fp=fopen(argv[1],"r");              /*以只读方式打开文件*/
13   if(fp==NULL)
14   {
15        printf("Open   file %s failed!", argv[1]);
16        return 2;
17   }
18   printf("Open file %s succeed!\n",argv[1]);
19   iflag=fclose(fp);                   /*关闭文件*/
20   if(iflag==0)
21        {
22             printf("Close file %s succeed!\n",argv[1]);
23             return 0;
24        }
25   else
26        {
27             printf("Close file %s failed! ", argv[1]);
28             return 3;
29        }
30    }
```

上述程序代码首先判断命令行输入参数是否满足要求，若不满足要求，输出提示信息后，程序退出(第 7~11 行)，如果满足要求，根据命令行中输入的文件名，用 fopen 函数以

只读方式打开文件(第 12 行)，并根据返回值判断文件打开失败还是成功，若不成功，输出错误提示信息后，退出(第 13~16 行)。如果文件打开成功，则输出打开成功信息之后再调用 fclose 函数将其关闭，并根据返回值情况判断文件关闭成功还是失败，如果关闭成功，输出成功提示信息后，程序退出，否则输出失败提示信息后，程序退出(第 18~30 行)。以下为程序运行结果：

```
$ ./ex1 test.txt
Open file test.txt succeed!
Close file test.txt succeed!
```

例 4-2　fcloseall 函数的使用方法。

```
1     /* ex2.c */
2     #include <stdio.h>
3     int main(int argc, char *argv[])
4     {
5             FILE *fp1, *fp2;
6             if((fp1=fopen("file1.txt","w"))==NULL) /*创建一个名为 file1.txt 的文件并打开 */
7             {
8                     printf("Open file1.txt failed!\n");
9                     return 1;
10            }
11      printf("Open file1.txt succeed!\n");
12            if((fopen("file2.txt","w"))==NULL)              /*创建名为 file2.txt 的文件并打开*/
13            {
14                    printf("Open    file2.txt failed!\n");
15                    return 2;
16            }
17      printf("Open file2.txt succeed!\n");
18            if(fcloseall()==EOF)                     /*关闭所有文件流*/
19                    {
20                            printf("close fil1.txt file2.txt failed!\n");
21                            return 3;
22                    }
23            else
24                    printf("streams closed succeed!\n");
25            return 0;
26    }
```

上述代码首先使用 fopen 打开 2 个文件，分别是 file1.txt 和 file2.txt，并判断文件打开是失败还是成功，若任一文件打开不成功，输出失败提示信息后，程序退出，否则输出打开成功提示信息(第 6~17 行)。之后通过 fcloseall 一次把全部文件流关闭，并判断文件关闭是否成功，若不成功，则输出失败提示信息后，程序退出，否则输出成功关闭信息后，正

常退出(第 18~26 行)。以下为程序运行结果：

```
$ ./ex2
Open file1.txt succeed!
Open file2.txt succeed!
streams closed successfully!
```

4.4 读 和 写 流

一旦打开了一个流，就能对流进行读写。既可以按无格式方法，也可以按有格式方法读写一个流。首先介绍无格式读写流的函数。有 3 种类型的无格式 I/O 函数可供选择：

- 字符 I/O 函数。这种函数每次读或写一个字符，由标准 I/O 库函数来处理缓冲。
- 行 I/O 函数。如果想每次读写一行，则可使用如 fgets 和 fputs 这样面向行的 I/O 函数。所读写的每一行是以换行符终止的。
- 块 I/O 函数。函数 fread 和 fwrite 支持块 I/O，它们每次读写若干个对象，每个对象的大小是指定的。这两个函数常常用于读写二进制文件，每次调用读写一个给定大小的数据结构。块 I/O 有时也称为二进制 I/O、对象 I/O 或结构 I/O。

4.4.1 字符 I/O

标准 I/O 库提供了一些支持基于字符的输入/输出函数，这些函数为验证某些字符串或对某个字符串中某个字符的提取提供了方便。以下 3 个函数允许一次读入一个字符：

```
#include <stdio.h>
int fgetc(FILE *fp);
int getc(FILE      *fp);
int getchar();
```

其中，参数 fp 为已打开的流结构指针。

函数 fgetc 从流 fp 中按 unsigned char 类型读取下一个字符，并将其值转换为 int 类型返回。若遇到文件结束或者出现读错误，则返回 EOF。

getc 的功能与 fgetc 相同，唯一不同的是允许将 getc 作为宏来实现，而 fgetc 则必须为函数。实际上，getc 常常是被高度优化了的，因此它是用于读取单个字符的最常用的函数。

getchar 等价于以 stdin 作为 fp 参数值的 getc，即 getc(stdin)。

这 3 个函数之所以将字符视为 unsigned，是为了保证在其高位被设置时函数的返回值不会为负值。之所以要求返回值为整型是为了能够返回所有可能的字符，包括文件结束和出现错误时的指示符 EOF，常数 EOF 的值常常为-1。这种表示意味着我们不能将 fgetc 的

返回值存储在字符类型的变量中，然后再将该值与常数 EOF 相比较。

与 3 个字符输入函数对应有如下 3 个字符输出函数：

```
#include <stdio.h>
int fputc(int c, FILE *fp);
int puts(int c, FILE *fp);
int putchar(int c);
```

fputc 将字符 c 转换为 unsigned char 类型，然后将它写至流 fp。若出现错误，它返回 EOF，否则返回字符 c。

类似于 getchar 函数，putchar 等价于 putc(stdin)，而 putc 则与 fputc 相同，但它常常是用较快的宏来实现的。putc 是用于输出单个字符的最常用的函数。

例 4-3 完成 Linux 中类似 wc 命令功能的程序。

```
1    /* ex3.c*/
2    #include <stdio.h>
3
4    #define BEGIN 1;                    /*开始读一个新单词，置为 BEGIN */
5
6    int main(int argc, char *argv[])
7    {
8    int c, characters, lines, words, state;    /*这些变量分别保存 getchar 函数的返回值、字符数、行数、
     单词数、记录是不是开始分析一个新单词的状态 */
9    /*设置初始值 */
10   state=0;
11   characters=words=lines=0;
12   /* 每次标准输入设备读入一个字符，直到输入字符‘0’ */
13   while((c=getchar())!='0')
14   {
15       characters++;          /* 单词数加一 */
16       switch(c)
17       {
18       case '\n':
19           lines++;           /* 行数加一 */
20           state=0;           /* 新行标志，表示单词的结束 */
21           break;
22       case ' ':
23           state=0;           /* 空格字符表示单词的结束*/
24           break;
25       case '\t':
26           state=0;           /* 制表符表示单词的结束 */
27           break;
28       default:               /* 其他情况，在某个单词中 */
```

```
29              if(state==0)
30                  {
31                      state=BEGIN;
32                      words++;        /* 如果是旧单词的结束，开始一个新单词*/
33                  }
34              break;
35          }
36      }
37      printf("There is %d characters, %d words, %d lines. \n",characters, words, lines); /*输出结果 */
38
39      }
```

Linux 中的 wc 命令的功能为统计指定文件中的字节数、字数、行数，并将统计结果显示输出。上述程序所实现的是类似的功能。调用 getchar 从标准输入设备读入一个字符(第 13 行)，分析这个字符，判断是不是新单词，新行，做一系列相应的操作，输出结果(第 15~37 行)。当然，这个 getchar 也可以改为“getc(stdin)”或“fgetc(sdin)”，程序的运行情况是一样的。程序运行结果如下：

```
$ ./ex3                                     /* 运行程序 */
one two linux ubuntu                        /* 输入单词 */
six
seven
desktop
tool
0                                           /* 输入单词结束 */
Thereis 44 characters, 8 words ,5 lines.    /* 输入结果 */
```

4.4.2 行 I/O

有许多应用是按行来处理数据的，例如编译程序通常就是每次读入一行源程序来进行词法扫描。标准 C 库中有两个函数用于每次读入一行：

```
#include <stdio.h>
char * fgets(char *buf, int count, FILE *fp);
char * gets(char *buf);
```

其中，参数 buf 是接受输入的缓冲区地址。count 是需要接收的字符数，fp 是流结构指针。

fgets 函数从 fp 指定的流中至多读入一行字符至参数 buf 指定的字符串中，参数 count 指明字符串 buf 的空间大小。该函数从流中连续读字符直至读到换行符或者读够 count-1 个字符(包括换行符)为止。所读入的这一行字符，包括最后的换行符，存储在参数 buf 指定的字符串中，并且在其末尾添加一个空字符(\0)作为该字符串结束标志。

如果要读入的这一行，包括结尾的换行符，长度大于 count-1，则只有部分字符被读入，但字符串 buf 仍以空字符结尾。下一次调用 fgets 将返回此行剩余的部分。

gets 函数不受参数 count 的限制，它从标准输入流 stdin 中读入完整的一行字符至参数 buf 指定的字符串中。它删除换行符并且在字符串尾部添加一个空字符。

如果调用这两个函数时文件已处在文件尾，则字符串 buf 的内容不被改变且返回值都是空指针。当它们遇到错误时，其返回值也是空指针。否则，返回值是指向字符串 buf 的指针。

每次输出一行可分别由 fputs 和 puts 两个函数来完成。

```
#include <stdio.h>
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

其中，参数 str 为要输出的字符串，fp 是流结构指针。

fputs 函数的作用是把以空字符(\0)结尾的字符串输出到某个特定的流中，末尾的空字符(\0)并不输出。由于字符串并没有要求一定以换行符结尾，所以这个函数虽然大多数情况下是一次输出一行的，但并不是必然的。

puts 函数的作用也是把以空字符(\0)结尾的字符串输出到标准输出设备上，同样不输出末尾的空字符(\0)，但是这个函数必定输出一个换行符。所以，puts 函数一定是一次输出一行的。由于使用这个函数不必处理换行符的问题，所以，它比 fputs 的应用更简单。

例 4-4　行输入输出函数的程序示例。

```
1     /* ex4.c */
2     #include <stdio.h>
3     int main(int argc , char *argv[])
4     {
5             char buf[1024];
6             FILE *fp;
7             if((fp=fopen("file1.txt", "r"))==NULL)
8             {
9                     printf("File open error. \n");
10                    return 1;
11            }
12            while((fgets(buf,1024,fp))!=0)
13                    puts(buf);
14            return 0;
15    }
```

这个程序十分简单。首先申请了一块 1024 个字符的字符串空间(第 5 行)；然后打开文件 file1.txt，判断是否成功，若不成功，报错退出(第 7~11 行)；若成功，接着一行行地从文件中输入，同样再一行行地输出到标准输出设备上，直到文件末尾，正常退出(第 12~14

行)。由于 puts 一定输出一个换行符，而本身 fgets 又将文件中的换行符读了进来，可以看到输出效果是隔行的。程序执行结果如下：

```
$ ./ex4                  /*运行程序 */
How are you?             /*程序运行结果 */

Fine.Thank you.

Linux Ubuntu

C programming
```

4.4.3 块 I/O

块 I/O 也称为二进制 I/O，它以固定大小的块为单位而不是以字符或行为单位来读写数据。所读写的数据既可以是字符正文，也可以是二进制数据。函数 fread 和 fwrite 用于进行这种成块的块的输入输出。

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t count, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *fp);
```

其中，参数 ptr 是指向若干个结构的指针，这个结构就是输入输出的最小处理单位；size 为结构的大小，一般用 sizeof 函数求得；count 是要处理的结构的个数；如为流的指针。

fread 从 fp 指定的流中读取 count 个数据项存放至 ptr 所指的数组中，其中每一项数据长度为 size 字节，所读取的总字节数为 count*size。

fwrite 从 ptr 所指的数组中写出 count 个数据项至 fp 指定的流，其中每一项数据长度为 size 字节，所写出的总字节数为 count*size。

fread 和 fwrite 均返回实际读写的数据项数(注意，不是字节数)。若调用成功，其返回值等于 count，若遇到文件尾或读错误，其返回值小于 count；当出现读错误时，设置流的错误指示器；如果 count 或 swize 为 0，则不做任何动作并返回 0。

这两个函数常在如下情形中使用：

(1) 读写一个二进制数组。例如，为了输出一个浮点数组的第 2 至第 5 个元素，可以这样调用 fwrite：

```
float data[10];
if(fwrite(&data[2],sizeof(float),4,fp)!=4)
 printf("fwrite error!\n");
```

此处指定参数 size 为数组元素的字节大小，参数 count 为元素个数。

(2) 读写一个结构。例如：

```
struct{
 int count;
 float score;
 char name[100]
} item;
if(fwrite(&item,sizeof(item),1,fp)!=1)
 printf("fwrite error");
```

此处指定参数 size 为结构的字节大小，参数 count 为 1。

这两种情形的更一般例子是读写一个结构数组。为此，参数 size 应当是该结构的字节大小，参数 count 应当是数组的元素个数。

例 4-5 下面的程序，将存放学生各种信息的文件中的学生信息读出，重新组成一个存放所有学生的前 3 门成绩的文件。

```
1    /* ex5.c */
2    #include <stdio.h>
3
4    #define NAMESIZE 30
5
6    struct {
7    char name[NAMESIZE];
8    long number;
9    short department;
10   short scores[10];        /*保存学生成绩的数组 */
11   } student;               /*保存一个学生信息的结构 */
12
13   short *pscores;          /*保存学生成绩的数组 */
14
15   int main(int argc, char *argv[])
16   {
17        FILE *fpstudents;                    /*已经存在的学生信息文件 */
18        FILE *fpscore;                 /*未存在的学生信息文件 */
19   /*判断命令行输入是否正确 */
20        if(argc<=2)
21        {
22              printf("usage: %s sourcefile destfile\n",argv[0]);
23              return 1;
24        }
25   /*打开学生信息文件，判断是否出错。*/
26        if((fpstudents=fopen(argv[1],"r"))==NULL)
27        {
```

```
28                printf("Open    sourcefile %s failed!", argv[1]);
29                return 2;
30          }
31          /*创建学生成绩文件，判断是否出错。*/
32          if((fpscore=fopen(argv[2],"w"))==NULL)
33          {
34                printf("Create    destfile %s failed!", argv[2]);
35                return 3;
36          }
37          /*将成绩前 3 项写入文件中  */
38          while(fread(&student,sizeof(student),1,fpstudents)==1)
39          {
40                pscores=student.scores;
41                if(fwrite(&pscores,sizeof(short),3,fpscore)!=3)
42                      printf("Error in writing file.\n");
43                return 4;
44          }
45          return 0;
46    }
```

在上面的程序中，首先定义了一个保存学生信息的结构(第 6~11 行)，然后从命令行输入已经存在的保存学生信息的文件和要创建的保存学生成绩的文件(第 20~24 行)，如果命令行输入正确，则打开学生信息文件，并判断打开是否成功，若不成功，输出错误提示信息后，退出(第 26~30 行)，成功后创建一个保存学生成绩的文件，若创建不成功，输出错误提示信息后，退出(第 32~36 行)，随后将学生成绩一一读取，并将成绩前 3 项写入文件中，如果写入过程中出错，输出错误提示信息后，程序退出(第 38~44 行)。

4.5　流文件定位

在读写一个文件之前，常常会需要移到文件的某个特定位置。例如，对于那种由若干固定大小的记录组成并且能用整数索引来引用这些记录的文件，为访问其中某个特定记录，最容易的方法是直接定位至该记录位置进行读写，而不必一个一个地顺序跳过该记录之前不需要的记录。为此，我们需要能够随意定位文件的位置，即随机地读写文件的任何部分。

标准 I/O 库提供了两种方法定位一个 I/O 流。

- ftell 和 fseek。它们假定文件位置是一个长整数。
- fgetpos 和 fsetpos。这 2 个函数比 ftell 和 fseek 有更好的兼容性。

下面先看一下 ftell 和 fseek 函数。

```
#include <stdio.h>
long int ftell(FILE *fp);
int fseek(FILE *fp,long int offset, int whence);
void rewind(FILE *fp);
```

其中，参数 fp 是流结构指针，offset 是流的偏移值，whence 指明参数 offset 的偏移起点。

ftell 函数调用成功返回 fp 所指定流的当前文件位置，它是从文件开始的字节数；否则返回-1。

fseek 函数能够改变 fp 所指流的文件位置。其中参数 whence 必须是表 4-2 所列之值，它指明参数 offset 的偏移起点，所允许的起点为文件开始、文件尾或者当前文件位置。参数 offset 给出相距 whence 指定位置的字节偏移量，它可以是正数，也可以是负数。

表 4-2　whence 参数取值选项

选　　项	说　　明
SEEK_SET	文件位置定位于文件开始+offset 之处
SEEK_CUR	文件位置定位于文件当前位置+offset 之处
SEEK_END	文件位置定位于文件尾+offset 之处

fseek 调用成功返回 0，失败返回非 0。如果调用成功，它清除流的文件结束指示器并忽略由 ungetc 退回的字符。如果该流是输出流并且缓冲的数据还未写至相连的文件，fseek 将导致未写出的数据被写至文件。因此，对于以更新方式(“+”)打开的文件而言，调用 fseek 之后，在此文件上的下一个操作既可以是输入，也可以是输出。

fseek 允许设置文件位置超过文件的当前文件尾，如果之后在此新文件位置写入了数据，则后续从原文件后与新写入的数据之间的空隙中读出的字节将用 0 填充直至此空隙写入实际的数据为止。

rewind 函数定位 fp 指定的流于文件的开始，它的作用等价于 fseek(fp,0L,SEEK_SET)；但有所不同的是返回值被忽略且重置了该流的错误指示器。

例 4-6　下面的程序说明了 fseek 的用法。

```
1    /* ex6.c */
2    #include <stdio.h>
3    struct record{
4    int uid;
5    char login[10];
6    };
7
8    char *logins[]={"user1","user2","user3","user4","user5"};
9    /*写出第 i 个位置上的记录 */
10   void putrec(FILE *fp, int i, struct record *ps)
```

```
11	{
12	/*  定位至文件的第 i 个记录位置处  */
13	fseek(fp,(long)i*sizeof(struct record),0);
14	fwrite((char *)ps,sizeof(struct record),1,fp);
15	}
16
17	int main(int argc, char *argv[])
18	{
19	int i;
20	FILE *fp;
21	struct record rec;
22	if(argc<=1)
23	{
24		printf("usage: %s datafile \n",argv[0]);
25		return 1;
26	}
27	/*新建数据文件  */
28	if((fp=fopen(argv[1],"w"))==NULL)
29	{
30		printf("Create    file %s failed!", argv[1]);
31		return 2;
32	}
33	/*  按逆顺序处理每一个用户，但输出的记录在文件中按实际顺序排列  */
34	for(i=4;i>=0;i--)
35	{
36		/*创建并输出该记录。参数 i 指定是第几个记录*/
37		rec.uid=i;
38		strcpy(rec.login,logins[i]);
39		putrec(fp,i,&rec);
40	}
41	fclose(fp);
42	return 0;
43	}
```

在上面的程序中，首先定义了一个结构用于保存用户的序号和用户名(第 3~6 行)。在函数 putrec 中用 fwrite 每输出一个结构成员之前，先调用 fseek 定位该元素在文件中的位置，这个位置是相对文件开始的位置，其偏移量为 i*sizeof(struct record)个字节，i 是数组元素下标。接着用 fwrite 函数/输出该记录(第 10~15 行)。在 main 函数中首先判断命令行输入是否符合要求，不符合要求的话，输出提示信息后，程序退出(第 22~26 行)。若符合要求，则创建一个新数据文件流，用于保存输出结果，若创建文件不成功，输出错误提示信息后，程序退出(第 28~32 行)。随后按逆顺序形成结构数组 logins 的各个元素并按此顺序将它们输出至流，但是输出的数组元素在文件中仍然按元素顺序排列(第 34~40 行)。所有

记录处理完成之后，关闭流，程序正常退出(第 41~42 行)。

接下来介绍 fgetpos 和 fsetpos 函数。这 2 个函数的格式如下：

```
#include <stdio.h>
int fgetpos(FILE *fp, fops_t *pos);
int fsetpos(FILE *fp, const fops_t *pos);
```

其中，参数 fp 是流指针，pos 为指向 fpos_t 的指针，pos_t 是一个存放指针位置的记录类型。

这两个函数也是定位流的操作，fgetpos 可以得到读写指针的位置，而 fsetpos 可以定位读写指针的位置。它们和前面介绍的 ftell 和 fseek 有些区别：

(1) fgetpos 和 fsetpos 使用了一种抽象的数据结构，fpos_t。在非类 UNIX 系统中，fpos_t 也可以被定义为存放文件读写位置信息的记录类型，所以这两个函数也可用于非类 UNIX 系统。

(2) ftell 和 fseek 中，读写指针的位置是用长整数来记录的，只能使用于类 UNIX 系统中。

这两个函数调用成功均返回 0，否则返回非 0。

例 4-7 以下程序说明了 fseek 和 fgetpos 函数的使用方法。

```
1   /* ex7.c */
2   #include <stdio.h>
3   #include <stdlib.h>
4   char buf[132];
5
6   int main(int argc, char *argv[])
7   {
8   FILE *fp;
9   fpos_t pos;
10  if(argc!=2)
11  {
12          printf("usage: %s mode \n",argv[0]);
13          return 1;
14  }
15    /*  打开一个文件*/
16  if(argv[1][0]!='a')
17  {
18          /*打开该文件写数据*/
19          if((fp=fopen("testfile","w+"))==NULL)
20          {
21                  printf("fopen failed!\n");
22                  return 2;
23          }
```

```
24   }
25   /* 打开该文件添加数据*/
26   else
27   {
28           if((fp=fopen("testfile","a+"))==NULL)
29           {
30                   printf("fopen failed!\n");
31                   return 3;
32           }
33   }
34    /* 写入 2 行数据*/
35   fputs("1234567890",fp);
36   fputs("abcdefghij",fp);
37    /* 查看当前文件尾位置 */
38   fseek(fp,0,SEEK_END);
39   fgetpos(fp,&pos);
40   printf("current file end position is %ld \n",pos);
41   /*定位至文件尾之后 30 字节 */
42   fseek(fp,30,SEEK_END);
43    /* 查看当前文件尾位置*/
44   fgetpos(fp,&pos);
45   printf("call fseek(fp,30,SEEK_END)\n");
46   printf("current file position is %ld \n",pos);
47    /*写入数据*/
48   fputs("abcdefg",fp);
49   printf("write %c %s %c \n",'\"',"abcdfg",'\"');
50   /*查看当前文件尾位置 */
51   fgetpos(fp,&pos);
52   printf("current position of file is %ld \n",pos);
53   fclose(fp);
54
55 }
```

上面的程序说明了 fseek 定位文件位置超过文件尾的两种情形。第一种是以非添加方式写数据超过文件尾，此时在当前文件尾和新写入的数据之间将形成所谓的“空洞”(第 42~48 行)。第二种是以添加方式写数据，此时，尽管可以用 fseek 定位文件当前位置超过文件尾，但是在写该文件时，文件的当前位置被忽略，所有数据都添加在文件尾并且文件当前位置被重新定位至新的文件尾(第 35、36 行)。为了明确给出文件位置，使用了 fgetpos 函数(第 39、44、51 行)。

如下所示运行这个程序：

```
$ ./ex7 a+    /* 程序名为 ex7 */
current file end position is 20
```

```
call fseek(fp,30,SEEK_END)
current file position is 50
write " abcdfg "
current position of file is 27
$ od -c testfile   /*查看 testfile 文件内容 */
0000000   1   2   3   4   5   6   7   8   9   0   a   b   c   d   e   f
0000020   g   h   i   j   a   b   c   d   e   f   g
0000033
$ ./ex7 w+        /*  打开文件写数据 */
current file end position is 20
call fseek(fp,30,SEEK_END)
current file position is 50
write " abcdfg "
current position of file is 57
$ od -c testfile    /*查看 testfile 文件内容 */
0000000   1   2   3   4   5   6   7   8   9   0   a   b   c   d   e   f
0000020   g   h   i   j  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000060  \0  \0   a   b   c   d   e   f   g
0000071
```

可以看到以添加方式写入的数据没有理会 fseek 定位的位置(此位置距文件开始为 50 个字节)，新写入的数据仍然紧接在当时的文件尾之后，即“abcdefghij”之后，此位置距文件开始 20 个字节。而以非添加方式(w+)写入的数据“abcdefg”则实际写在由 fseek 定位的位置，即当时的文件尾 30 个字节之后(第 50 个字节，注意，od 命令输出中的第一列给出的是八进制数字)，在它们之间有 30 个 null 字符。

4.6　文件结束和错误

本章描述的许多函数(fgets，gets，putc，ungetc，fread 等)返回 EOF 指明操作未成功完成。由于 EOF 既用于报告文件结束也用于报告随机出现的错误，因此，为了区分究竟是错误返回还是文件结束返回，有时还需要调用 ferror 函数来确定是否存在错误，调用 feof 函数检查是否遇到文件结束。

每一个流对象内部都保持着两个指示器：一个为错误指示器，当读写文件出错时该指示器被设置；另一个为文件结束指示器，当遇到文件尾时该指示器被设置。ferror 和 feof 两个函数分别对这两个指示器进行检查。

```
#include <stdio.h>
int ferror(FILE *fp);
```

```
int feof(FILE *fp);
void clearerr(FILE* fp);
```

ferror 函数返回 1 当且仅当 fp 所指流的错误指示器被设置，否则返回 0。

feof 函数返回非零值，当且仅当 fp 所指流的文件结束条件指示器被设置，否则返回 0。

clearerr 函数消除这两个指示器。

例 4-8 下面的程序说明了 ferror、feof 和 clearerr 函数的使用方法。

```
1     /* ex8.c */
2     #include <stdio.h>
3
4     int main(int argc, char *argv[])
5     {
6       int i;
7       FILE *fp;
8       if(argc<=1)
9       {
10          printf("usage: %s file \n",argv[0]);
11          return 1;
12      }
13      /* 以写方式新建一个文件 */
14      fp=fopen(argv[1],"w");
15      /* 从该文件中读取 1 个字符 */
16      fgetc(fp);
17      printf("%d \n",ferror(fp));
18      /* 往文件中写入字符 */
19      fputs("abcdefgh",fp);
20      /*关闭该文件流再重新打开 */
21      fclose(fp);
22      fp=fopen(argv[1],"r");
23      /* 设置流指针到文件尾 */
24      fseek(fp,0,SEEK_END);
25      fgetc(fp);
26      if(feof(fp))
27      printf("file end\n");
28          /* 清除指示器 */
29      clearerr(fp);
30      printf("%d %d\n",ferror(fp),feof(fp));
31      fclose(fp);
32      return 0;
33    }
```

上面的程序首先建立一个新文件，并用 fgetc 函数读取一个字符，由于此时文件为空

文件，因此无法读取，调用 ferror 函数后，返回(第 14~17 行)。然后往新建的文件中写入一些字符，关闭该文件流再重新打开，并设置流指针到文件尾(第 19~24 行)。此时调用 feof 函数后，返回值为 TRUE(第 26、27 行)。再调用 clearerr 函数后，这个指示器被清除，因此再次调用 feof、ferror 后，函数返回值为 0(第 29、30 行)。程序执行结果如下：

```
$ ./ex8 aaa          /*执行程序 */
1
file end
0 0
```

4.7 流 缓 冲

写到一个流的字符并不是一旦执行输出函数就立即写到文件中，而是先在一个缓冲区中聚集为一块，然后异步地以块为单位传送到文件。类似地，从一个流读出的字符也不是一个一个地从文件中读出的，而是以块为单位从文件中读出的。这种处理方式称为缓冲。

采用缓冲的目的是为了减少调用低级 I/O 函数 read 和 write 的次数，因为这些真正读写文件的函数是系统调用，它们是十分费时间的操作。例如，对于存储在硬盘上的文件而言，当进程用 read 或 write 读写某个具体数据时，设备驱动程序必须将数据在文件中的地址转换成硬盘的物理磁道号、卷宗号以及扇段号。然后设备要移动磁头至相应的卷宗并等待磁盘片相应扇段旋转至磁头之下，这些都是费时的机械动作。一切准备好了之后才能从磁盘开始读写数据。显然，每读写一个或几个字符便导致这一串的机械动作是极不合算的。利用缓冲处理则不必为每读写一个字符而频繁地与外部设备打交道，同时还可以实现异步 I/O。即在 CPU 运行程序的同时从外设传输数据，从而提高输入输出的效率。

标准 I/O 库函数自动地管理缓冲区，这使得我们无需过问何时该从文件中读一块数据至缓冲区，当前缓冲区中字符的位置以及一些与特定设备有关的事情，如对终端设备的控制操作等细节问题。

流有 3 种不同的缓冲类型：

(1) 全缓冲。在这种情况下，实际 I/O 操作每次读写的数据是以整个缓冲区为单位的。对于输出，只有当流缓冲区满了时才传送它至文件；对于输入，每次从文件读入数据直至缓冲区满为止。驻存在磁盘上的文件正常情况下是全缓冲的，所采用的缓冲区由 I/O 标准函数在第一次对流进行 I/O 操作时用 malloc 分配。

(2) 行缓冲。在这种情况下，标准 I/O 库仅当在输入或输出中遇到换行符时才执行实际 I/O 操作。行缓冲通常用于诸如终端之类交互设备的流。例如，如果我们用 fputc 输出 15 个非换行字符，然后输出一个换行符，则只有当输出换行符的这个 fputc 被调用后，前面输出的 15 个字符才能真正出现在终端上。

(3) 无缓冲。流不设置缓冲区，从流中读出或写入至流的字符单个单个地从文件传出或被传送至文件。标准错误流通常是无缓冲的，这是为了使得错误信息及时被显示出来。这意味着如果我们用 fputc 输出 15 个字符至代表错误流的终端，则每一个字符都将在函数被执行后立即出现在终端上。

在 Linux 系统中，对新打开的流采用如下默认缓冲类型：

- 标准错误流总是无缓冲的。
- 其他的流若引用交互设备则是行缓冲的，否则是全缓冲的。

这种自动默认选择通常给予所打开文件或设备一种最方便的缓冲。不过，如果我们不满意这种默认缓冲的话，也可以用如下函数设定自己希望的缓冲类型和缓冲大小。

```
#include <stdio.h>
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
void setbuf(FILE *fp, char *buf);
void setbuffer(FILE *fp, char *buf, size_t size);
void setlinebuf(FILE *fp);
```

这些函数中的 fp 是已经打开的流，buf 是用户自己设定的缓冲区，用于替换系统默认的缓冲，size_t size 是缓冲区的大小，mode 是流的类型，可以取_IOFBF、_IOLBT 和_IONBF，分别代表全缓冲、行缓冲和无缓冲，它们都是定义在<stdil.h>中的常数。调用这些系统函数时，必须已经打开了流，这样才有 fp 指针存在。这些调用可以在任何时候作用于流，来改变流的缓冲区，但要求这时的流是非“活跃”的，即没有进行 I/O 操作之前，或刚刚执行了一个 fflush 之后。当然，最好还没有进行其他的操作，因为其他的操作是和缓冲的性质密切相关的。

如果指定无缓冲类型，则 setvbuf 将忽略参数 buf 和 size；否则根据 buf 和 size 指定缓冲区及其大小。

如果用 NULL 作为 buf 的值，则 setvbuf 会自动地为此流分配适当大小的缓冲区。所谓适当大小是指与此流相连文件的 stat 结构成员 st_blksize 指定的值。如果系统不能为流确定这个值(例如，当流与设备或管道相连时)，则分配 BUFSIZ 长度的缓冲。BUFSIZ 是定义在 stdio.h 中的常数，它的值至少为 256。当流被关闭时，这样分配的缓冲区将被自动释放。

buf 应当是一个字符数组，它至少应当能够容纳 size 个字符。setvbuf 使用该数组作为流缓冲区并释放标准 I/O 库原来分配的缓冲区。对于这个数组我们应当注意以下两点：

(1) 只要流是打开的，就不能释放该数组的空间。通常应当要么静态地分配此数组，要么用 malloc 函数为它分配空间。用自动数组作为缓冲区是不好的，除非在退出说明该数组的程序块之前关闭文件。

(2) 流 I/O 函数将使用这个数组用于内部目的。当流正用它作为缓冲目的时，我们不能直接访问该数组的内容。

setvbuf 函数调用成功返回 0；否则返回非 0 表示出错。

setbuf 函数实际上是 setvbuf 的特例，当 buf 是 NULL 时它等价于：

```
setvbuf(fp, buf, _IONBF, BUFSIZ);
```

当 buf 是非空指针时，它等价于：

```
setvbuf(fp,buf, _IOFBF, BUFSIZ);
```

setbuf 函数用于打开或关闭 fp 指定流的缓冲。为了打开缓冲，参数 buf 必须指向一个长度为 BUFSIZ 的缓冲区。通常在此函数调用之后流为全缓冲流，但如果流是与终端设备相连的话，则有的系统设置为行缓冲流。为了关闭缓冲，参数 buf 必须是 NULL。

setbuffer 这个函数与 setbuf 比较一致，只是在 buf 不为空时，设定的全缓冲的大小不是系统预定值，而是 size。

setlinebuf 这个函数是专门用来将缓冲区设定为行缓冲的。

术语刷新表示写出缓冲区中的数据。缓冲区可以由标准 I/O 库函数自动地刷新，也可以通过调用 fflush 函数来刷新。通常，缓冲区中的数据在下述情况下会自动刷新：

- 当流被关闭时。
- 当调用 exit 终止程序时。
- 若流是行缓冲的，当写出一换行符时。
- 当企图输出而缓冲区已经满了时。
- 无论何时对流的输入操作导致它实际从文件读数据时。

例如，在多数系统上，行缓冲区的大小通常是固定的。因此，如果在输出换行符之前一次输出的字符太多以致缓冲区满了时，尽管还未输出换行符，系统也会自动地将缓冲区中的内容刷新。这是上述第 4 种情形的例子。第 5 种情形的一个例子是：当用 printf 输出不带换行符的一个字符串至终端之后，若紧接着调用从终端读数据的函数，则也会导致缓冲区的输出立即被写到终端。这就是为什么用 printf 输出不带换行符的字符串时，有时候该字符串能立即出现在终端上(因为其后跟有输入操作)，而有时候却必须使用 fflush 函数才行的原因(见程序 9)。

如果想在其他时刻将缓冲区的内容刷新，就要显式地调用 fflush 函数。

```
#include <stdio.h>
int fflush(FILE *fp);
```

fflush 导致刷新 fp 所指流上的缓冲，即导致缓冲区中还未写出的输出被传送至其文件。如果 fp 是一空指针 NULL，则 fflush 刷新所有打开的输出流上的缓冲。

fflush 调用成功返回 0，否则返回 EOF。

虽然标准 I/O 库函数自动地为我们管理 I/O 缓冲区，但标准 I/O 库中最让人感到迷惑、也最简单的问题常常是由缓冲引起的。例如，如果编写的是一个使用流进行输入输出的程序，当设计它的用户界面时，就必须了解流缓冲是怎样工作的。否则的话，可能会发现输出(如程序进展显示或提示性的消息)并不像所预期的那样，甚至出现其他未曾料到的行为。

例 4-9　这是一个未注意到缓冲的作用而导致输出行顺序不对的例子。

```
1     /* ex9.c */
2     #include <stdio.h>
3
4     int main()
5     {
6         int c, answer;
7         printf("1: This is a buffer test program.\n");
8         /* fflush(NULL) ; */
9         fprintf(stderr,"2: Ihope this should be second line.\n");
10        printf("3: Hello, Are you a student?");
11        while(1)
12        {
13         c=tolower(fgetc(stdin));
14         answer=c;
15         while(c !='\n'&&c!=EOF)
16              c=fgetc(stdin);              /* 忽略该行后面的字符 */
17         if(answer=='y')                   /* 如果是回答字符，响应回答 */
18         {
19              printf("Hope you have good score.");
20              break;
21         }
22    else if(answer=='n')
23         {
24              printf("Hope you have good salary.\n");
25              break;
26         }
27         else                                   /* 非回答字符，请求合法回答 */
28              printf("Please anser y or n: ");
29        }
30        /* fflush(NULL) */
31        fprintf(stderr,"n-1: I hope this line should last but one.\n");
32        printf("\nn: Test over .\n");
33        return 0;
34    }
```

这个程序简单地提出一个问题，获取回答，然后对回答作出反应。为了显示输出被缓冲的情况，我们故意在第 7 行输出之后和最后 31 行输出之前加入了往标准错误流输出的语句。因为标准错误流是无缓冲的，对它的输出将立即出现在终端上。运行这个程序有如下结果：

```
$ ./ex9              /*执行程序 */
1: This is a buffer test program.
2: Ihope this should be second line.
```

```
3: Hello, Are you a student?i                    /* 输入字符 i */
Please anser y or n: y
n-1: I hope this line should last but one.
Hope you have good score.
n: Test over .
```

这种结果并不是我们期望的，我们原本希望的是按程序执行顺序输出每一行，即，以“1：”开头的应当输出在第一行，以“n-1”开头的应当输出在倒数第二行。为了保证这一点，应当要么在输出中加入适当的换行符，要么在程序适当位置加入 fflush 调用，即去掉程序中对 fflush 调用的注释。再重新编译运行，可以发现程序运行结果与我们的预期一致。

对于指明了“+”而打开的文件既可以读也可以写，不过，对这种文件，在从读转变为写或者从写转变为读时，必须调用 fflush，否则可能会由于缓冲的原因出现意想不到的错误。

例 4-10 下面的程序创建一个文件并写入两行数据之后关闭它，然后以读写方式再次打开该文件，并按读、写、读的顺序连续读一行和写一行数据。我们故意保留了两个被注释的 fflush 调用，正确的程序应当有这两个调用。由于在读、写和写、读中间没有调用 fflush，使得读写文件出现了奇怪的结果。

```
1       /* ex10.c */
2       #include <stdio.h>
3       /* 该函数的作用是从流中读入一行，并判断是否到达文件尾 */
4       int get_line(char *buf, int bufsize, FILE *fp)
5       {
6         if(fgets(buf,bufsize,fp)==NULL)
7         {
8               if(feof(fp))
9               {
10                      printf("End of file \n");
11                      return EOF;
12              }
13              else
14              {
15                      printf("fgets failed\n");
16                      return 0;
17              }
18        }
19        printf("call fgets: %s    \n", buf);
20        return 1;
21      }
22
23      char buf[132];
24      int main(int argc, char *argv[])
```

```
25          {
26            FILE *fp;
27            if(argc<=1)
28            {
29                printf("usage: %s file \n",argv[0]);
30                return 1;
31            }
32            /* 创建文件并写入 2 行数据 */
33            if((fp=fopen(argv[1],"w+"))==NULL)
34            {
35                printf("fopen failed!\n");
36                return 2;
37            }
38            fprintf(fp, "This is first line.\n");
39            fprintf(fp, "This is seconf line. \n");
40            fclose(fp);
41            /* 再次打开该文件读写数据 */
42            if((fp=fopen(argv[1],"r+"))==NULL)
43            {
44                printf("fopen failed!\n");
45                return 3;
46            }
47            /*读一行数据 */
48            get_line(buf,sizeof(buf),fp);
49            /* 由读变为写 */
50          /* fflush(fp); */
51            fprintf(fp, "This line should be the new second line.\n");
52            /* 由写变为读 */
53            /*fflush(fp); */
54            get_line(buf,sizeof(buf),fp);
55            fclose(fp);
56            return 0;
57          }
```

对上面的程序编译链接后，运行它，得到以下结果：

```
$ ./ex10 testfile10       /* 运行程序 */
call fgets: This is first line.                    /* 第一次 get_line 的输出 */
 End of file                                           /* 第二次 get_line 的输出 */
$ more testfile10
This is first line.
This is second line.
```

可以看到第二次调用 get_line 遇到了文件尾，并且文件 testfile10 中只有第一次创建文

件时写入的两行数据，而第二次打开新写入的数据不在文件中!

如果去掉程序中第二个 fflush 的注释(第 53 行)，重新编译并运行该程序，将得到：

```
$ ./ex10 testfile10        /* 运行程序 */
call fgets: This is firlst line.                    /* 第一次 get_line 的输出 */
 End of file                                          /* 第二次 get_line 的输出 */
$ more testfile10
This is firlst line.
This is second line.
This first line.
This line should be the new second line.
```

文件中虽然加入了新写的这一行，不过却保留了原来写入的第二行，同时还莫名其妙地多出了原来第一行的数据。如果仅去掉第一个 fflush 的注释后执行该程序(第 50 行)，则有：

```
$ ./ex10 testfile10        /* 运行程序 */
call fgets: This is first line.                     /* 第一次 get_line 的输出 */
call fgets:                                         /* 第二次 get_line 的输出 */
$ more testfile10
This is first line.
This line should be the new second line.
```

这一次虽然将新数据写入到了文件中，但第二个 fgets 却仍然返回空行。只有当同时去掉两个 fflush 的注释时，该程序才会得到预期的结果：

```
$ ./ex10 testfile10
call fgets: This is first line.
End of file
$ more testfile10
This is first line.
This line should be the new second line.
```

这个例子说明，当使用流 I/O 从读切换为写或从写切换为读时，为保证文件读写的可预期性，均应当调用 fflush 刷新流缓冲区。

4.8 格式化 I/O

前面几节描述的 I/O 函数除了将数据分解成字符或者行之外，并不对所操作的数据进行解释，但有时对数据进行解释却是必要的。我们知道数据在计算机内的表示与人们习惯的阅读形式不同。例如，十进制数 10 在计算机内部的 32 位表示是：

00000000000000000000000000001010

但是，当这个数在打印机上输出或者在终端屏幕上显示时，它必须转换为 ASCII 字符“1”和“0”，这 2 个字符在计算机内部有完全不同的表示：

1：00110001

0：00110000

类似地，为了从键盘读入十进制整数 10，人们习惯的十进制数表示必须转换为计算机可处理的内部表示。

格式化 I/O 函数能够自动完成这种数据外部格式和内部格式的转换工作，并且能够对输入输出数据进行诸如数据类型、精度、位置等格式控制，它们是标准 I/O 库中使用最频繁的函数。

所有格式化 I/O 函数的调用界面都是简单的，它们都通过一个格式字符串来对其余参数进行格式描述。但是，格式字符中的转换区分符则由于格式本身的复杂性而五花八门，因为它们要描述每一种数据类型(整数、浮点数、十进制数、八进制数、十六进制数、字符、字符串…)，要描述数据的精度(单精度、双精度、短整数、长整数…)，要描述数据的外部形式(指数形式、定点形式、左对齐、右对齐、有否前缀 0、正号或负号…)以及字节宽度等。详细描述所有这些转换区分符需要较大的篇幅，因此，我们这一节只描述这些函数的用法，简单列出转换区分符的语法成分，并通过例子说明它们的语法，关于转换区分符的更详细描述可以查阅联机手册。

4.8.1　格式输出

格式输出由 3 种 printf 函数来处理。printf 函数是完成输出最方便、最简洁的手段。

```
#include <stdio.h>
int printf(const char *format,…);
int fprintf(FILE* fp, const char *format, …);
int sprintf(char *buf, const char *format,…);
```

其中，参数 format 是格式参数，通过它来确定以什么方式输出数据；…是一些不定数目的参数，也就是要输出的数据，这些数据的个数必须与格式参数的个数相匹配。

这 3 个函数功能相同，它们转换字符串 format 并在 format 的控制下打印其他参数。若调用成功它们均返回输出的字符个数，失败则返回一个负数。不同的仅仅是输出的流不同；printf 输出至标准输出流；fprintf 输出至参数 fp 指定的流；sprintf 不是输出至一个文件，而是输出格式化的字符至参数 buf 所指的字符数组中，并且在 buf 的末尾自动添加一个空字节。

这些函数允许任意个数的参数，但都必须指定一个格式字符串 format。格式字符串指明后继的参数个数以及怎样解释和格式化这些参数，它由两类字符组成：转换区分符和普通字符。

普通字符是除转换区分符之外的其余字符。在格式字符串中的普通字符简单地按原样写至输出流。转换区分符是以“%”打头、按一定语法规则组成的连续字符。转换区分符导致 format 之后对应的参数被格式化后写至输出流。如果转换区分符的个数少于后继的参数个数，则多余的参数被忽略；如果转换区分符的个数多于后继的参数个数，则多余的转换区分符输出的内容是不确定的，如例 4-11 所示。

例 4-11　转换区分符应用示例

```
1    /* ex11.c */
2    #include <stdio.h>
3    int main()
4    {
5    int i=23;
6    char filename[]="file.txt";
7    printf("Processing of '%s' is %d %% finished. Please be patient.\n",filename, i);
8    printf("%s \n",filename,i);
9    printf("%s is finished. %d %s \n",filename);
10   return 0;
11   }
```

这个例子中格式字符串内以“%”打头的，如“%s”、“%d”、“%%”均是转换区分符，其中“%s”指明字符串参数的打印，“%d”指明 int 参数应按十进制表示打印出来，而“%%”表示打印一个‘%’字符。除此之外的字符均是普通字符。这 3 个 printf 调用将产生如下输出：

```
Processing of 'file.txt' is 23 % finished. Please be patient.
file.txt                    /*  参数 i 被忽略  */
file.txt is finished. 23 □Ö /*后 2 个值是不确定的随意值  */
```

转换区分符 “%s”、“%d”中，“%”之后的字符“s”和“d”称为转换字符，在第三章中已简单介绍过。转换字符在转换区分符中起关键的格式转换作用，它们用于指明参数的类型和打印的基本格式。在多数情况下只需要使用这种由单个转换字符组成的简单转换区分符就足够了。表 4-3 列出了所有输出转换字符。

表 4-3　输出转换字符

转 换 字 符		说　　明
整形转换	d, i	打印整数为有符号十进制数。d 和 i 对于输出而言是相同的，但用于 scanf 中输入时则不同(参见输入转换区分符表)
	o	打印整数为无符号八进制数
	u	打印整数为无符号十进制数
	x,X	打印整数为无符号十六进制数，x 用小写字母，X 用大写字母

(续表)

转换字符		说明
浮点转换	f	按正常的定点表示打印浮点数。产生形如[-]DDD. DDD 的输出，其中小数点后的数字个数受指定的精度控制。未指明精度时为 6 个数字
	e,E	e 转换字符按指数形式打印浮点数，产生形如[-] D.DDDe+DD 形式的输出。小数点后的数字个数受指定的精度控制。指数总是至少包含两个数字，E 转换字符类似于 e，但指数用字母 E 标志
	g,G	用定点形式或指数形式打印浮点数。采用哪一种形式更为适合则取决于参数值：当指数小于-4 或大于等于其精度时，按 e 或 E 格式打印；否则按 f 格式打印。在结果小数部分尾部的零将被去掉且仅当小数点之后有数字时才出现小数点
其他转换	c	打印单个字符。可用‘-’标志指明在域中左对齐，但没有其他标志，也没有精度成长度修饰符
	s	打印一字符串。对应参数必须满足 char *类型。可以用精度指明要写出的最大字符个数；否则，写出字符串中除终止空字符之外的所有字符。可用‘-’标志指定在域中左对齐，但没有其他标志或长度修饰符
	p	打印一指针。对应的参数必须是 void *类型，实际上可以使用任何类型的指针。允许用‘-’标志指定在域中左对齐，但没有其他标志、精度成长度修饰符
	n	获取迄今已打印的字符数。与它对应的参数必须是指向 int 的指针。注意，这种转换字符不会输出任何参数，它只是存储已打印的字符数与对应参数。允许用 h 和 l 长度修饰符指明参数的类型是‘short int *’或‘long int *’来替代‘int *’，但不允许标志、域宽或精度
	%	打印字符%。这个转换不使用参数，也没有标志、域宽、精度和长度修饰符

为了更精确地对格式进行控制，在“%”和转换字符之间也可以写上各种修饰符。例如，可以指定最小域宽度，对于浮点数也可以指定小数点后保留的位数，还可以用一个标志指定结果在此域中左对齐。例如：

```
float pi=3.1415926;
printf("pi=%10.5f \n",pi);
```

将输出：

```
pi=3.14160;
```

标准 I/O 库中还有另外 3 个格式输出函数 vprintf，vfprintf，vsprintf，它们与上述 3 个函数类似，唯一不同之处仅在于用可变参数指针数组 arg 替代了输出变量参数表。

4.8.2 格式输入

格式输入由如下 3 个 scanf 函数来完成：

```
#include <stdio.h>
int scanf(const char *format,...);
int fscanf(FILE *fp, const char *format, ...);
int sscanf(char *s, const char *format, ...);
```

scanf 从标准输入流读数据，fscanf 从 fp 指定的流读数据，sscanf 不是从文件而是从参数 s 指定的字符数组中读数据。每一个函数读取若干字节，根据 format 参数解释它们并存储结果于对应的参数中。位于 format 之后的可选参数应当是指向存放接收输入值的变量指针。这些函数的返回值正常情况下为成功读入了值的参数个数；如果在任何匹配或转换被执行之前检测到了文件结束条件，则返回 EOF。如果输入时出错导致设置流的错误指示器，则返回 EOF。

这一组 scanf 函数除了是从输入流读取数据并存储值至对应的参数所指的地址中之外，与 printf 函数族的工作方式十分类似，它们也用类似的方式使用格式字符串 format 控制输入转换，并且许多转换区分行也是相同的。但它们与 printf 函数族有两点重要的不同：

(1) 输入转换区分符的语法虽然与 printf 函数中的语法类似，但它们的解释主要是面向自由格式的输入和简单的模式匹配，而不是针对格式化的固定域。例如，大部分 scanf 转换都跳过输入文件中的空白字符(包括空格符、制表符、换行符等)，并且对于数值转换没有对应输出转换那样的精度概念。此外，输入格式字符串中的普通非空白字符预期精确地匹配输入流中的字符，但当匹配失败时并不认为是流输入错误(匹配失败不设置流的错误指示器)。

(2) scanf 与 printf 之间的另一个不同是：scanf 中位于 format 参数之后的所有其他参数都应当是指针；所读入的值将存储在指针所指对象之中。记住这一点非常重要，即使是有经验的程序员偶尔也会忘记这一点。

scanf 使用的格式字符串 format 由 3 类成分组成：一至多个连续的空白符(空格符“ ”、制表符“\t”、水平制表符“\v”、换行符“\r”或者走纸符“\f”，即 isspace 函数返回值为真的字符)，普通字符(不包括“%”和空白符)和转换区分符。

格式字符串中的空白符导致 scanf 跳过输入流中的空白字符，直至遇到一个末读过的非空白字符或者遇到文件尾为止。输入流中的空白字符不必完全与格式字符串中的空白符相同。例如格式字符串“，”识别输入流中一个前后有任意空白的逗号。

请注意区分术语“空白符”、“空格符”的“空字符”。空白符包括空格符“ ”、制表符“\t”、水平制表符“\v”、换行符“\r”和走纸符“\f”，即 isspace 函数返回值为真的字符；空格符只指“ ”；空字符是 null(即“\0”)字符。

格式字符串中的普通字符用于指明必须出现在输入流中的字符，它必须完全与输入流中的下一字符相匹配；如果不匹配将导致匹配失败。例如：

```
int num;
scanf("how %d", &num);
```

仅当在标准输入流中后续 3 个字符是“how”时，scanf 调用才会成功。在此之后，如果后面的字符形成了一个可识别的十进制数，则该数将被读出并存储在变量 num 中。这意味着对如下输入，scanf 调用将成功，并赋值 123 给 num：

```
how   123
how123
```

大部分转换区分符通常都忽略输入中的空白字符，这意味着%d 将一直读输入直至发现一个数字序列。如果所期望的字符没有出现，该转换将失败且 scanf 将立即返回。

格式字符串中的转换区分符指导下一个输入域的转换。输入域定义为输入流中非空白字符组成的字符序列，其长度直至遇到一个不合适的字符或者到达指定的域宽为止。转换后的结果存储在对应的参数中，除非转换区分符指明了禁止赋值标志“*”。

一般地，每一个输入转换区分符具有如下形式：

```
[posp $][*][width][size]fmt
```

表 4-4 和表 4-5 列出了各种输入转换字符和输入转换区分符的各种修饰符。

表 4-4　输入转换字符

选　　项	说　　明
d	匹配一个十进制形式的有符号整数
i	匹配一个 C 语言为指定一整常数而定义的任何形式的有符号整数
o	匹配一个八进制形式的无符号整数
u	匹配一个十进制形式的有符号整数
x,X	匹配一个十六进制形式的无符号整数，x 用小写字母，X 用大写字母
f,e,E,g,G	匹配一任意的有符号浮点数，它们之间可以互相替换
c	匹配单个字符或多个字符组成的字符串，读入的字符个数由最大域宽指示符指定或默认为 1。它不在读入的正文之后附加一空字符，也不跳过打头的空白符。它严格地读域宽给定的 n 个字符，并且当不能达到这么多个字符时将导致失败
s	匹配一个由空白字符组成的字符串。它跳过并丢弃打头的空白字符，仅在读入非空白字符之后停止于遇到的第一个空白字符。它在读入的正文尾部附加一空字符

(续表)

选　　项	说　　明
[	这种转换包括所有后续字符直至相匹配的右方括弧“]”。它匹配一个由特定字符集中的字符组成的字符串。这个特定字符集使用与正则表达式相同的语法定义于“[”和“]”之间。但有如下特殊情形： 如果这个特定集合中也包括字符“]”，则它必须是紧跟在初始“[”之后的第一个“]”字符，与初始“[”相匹配的右方括弧将是下一个“]”。 嵌入在正则表达式内的字符“-”(既不是第一个字符，也不是最后一个字符)用于指明字符的范围。 紧跟在初始“[”之后的脱字符“^”指明所允许的字符集合是除列出的字符之外的任何字符
p	用于读一指针值。它所识别的语法同 printf 的%p 输出转换相同，即一个恰如%x 转换接收的十六进制数。对应的参数必须是 void **类型，即一个存放指针的地址
n	这一转换不读任何字符，它记录此次调用中迄今为止已读入的字符数
%	匹配输入流中的字符%，这个转换不使用参数

表 4-5　输入转换修饰符

修　饰　府	说　　明
posp $	posp 为一个范围在[1~NL_ARGMAX]的十进制正整数，它指出参数表中下一个要输出的参数位置(紧接在 format 参数之后的那个参数的位置为 1)。这个修饰符使得格式字符串能够按任意顺序选择要赋值的参数和多次对同一参数赋值
*	禁止赋值标志。它使得 scanf 忽略所读出的输入值，但不使用指针参数并且也不增加成功赋值计数
width	一个十进制正整数，称为宽度指示符，用于指明最大域宽。当到达此最大域宽或发现一非匹配字符时，不论谁先发生，均停止读输入流。大多数转换字符均忽略打头的空白字符，这些被忽略的空白符不计数在此最大域宽内。字符串转换存储一空字符作为该输入的结束，最大域宽也不包括这个终止符
size	称为长度修饰符，指明接收对象的类型长度。当没有长度修饰符时，对应的参数处理成 int 类型(对于转换 i 和 d)，unsigned int 类型(对于转换 o，u，x 和 X)，或者 float 类型(对于 e，E，f，g 和 G 转换)。当接收对象的类型长度与默认类型长度不同时需要使用如下长度修饰符： h　对于 i，d 和 n 转换，指明参数是一个 short int *；对于 o，u 和 x 转换，指明参数是一个 unsigned short int *。 l　对于 i，d 和 n 转换，指明参数是一个 long int *；对于 o，u 和 x 转换，指明参数是一个 unsigned short int *。 对于浮点转换，指明参数是一个 double *。 L　对于浮点转换，指明参数是一个 long double *

同输出转换一样，输入转换也可以指明域宽限制输入吸收的字符数，可以指定长度修饰符指明接收输入值的参数类型长度；同样，%n $打头的输入转换也可以指定将输入结果存放在第 n 个参数所指对象中而不是下一个参数对象中；禁止赋值标志“*”可以描述要被跳过的输入。下面是一些使用输入转换区分符的例子：

(1) 对于输入“Hello,world”(注意，在逗号和单词 world 之间有一个空格)，用转换%10c 读将产生“Hello, wor”，它只读入 10 个字符，其结尾没有空字符；但用%l0s 读却产生“Hello, \0”，它停止在第一个空格字符处并在尾部添加空字符。

(2) 如下调用：

```
int i,n; float x; char name[50];
scanf("%d %f %80s", &i, &x, name);
```

对于输入“25 54.32E-1 Jack”，将赋值 3 给变量 n，赋值 25 给变量 i，赋值 5.432 给变量 x，而 name 中则将包含字符串“Jack \0”。

(3) 如下调用：

```
int i,n; float x; char name[50];
(void) scanf("%sd %f %*d %49[0-9] %n", &i, &x, name, &n);
```

对于输入“56789 0123 56a72”，将赋值 56 给 i，赋值 789.0 给 x，跳过 0123，并放置字符“56\0”于 name，读入的字符个数赋给变量 n，其值为 13。下一个待读入的字符是 a。需要注意的是，%n 是确定文字匹配成功与否的唯一手段。如果%n 位于匹配失败点之后则不会有值存入其内，因为 scanf 在处理%n 之前就已返回。如果在调用 scanf 之前先存储-1 至%n 对应的那个参数，则在调 scanf 之后若其值仍为-1 就表明到达%n 之前已遇到错误。

(4) 转换区分符“%2[ ][ ]”表示匹配至多由两对方括弧组成的字符串；“%16[a-z]”表示匹配由至多为 16 个小写字母组成的字符串；“%25[^\f \n \r \t \v]”表示匹配至多 25 个字符长度的字符串，但不包括任何标准空白字符。需要注意的是“%[”转换与“%s”稍有不同。因为如果输入以空白打头，“%[’报告匹配失败，而“%s”则简单地丢弃打头空白字符。此外还有一点需提醒的是，“%s”和“% [”转换若没有指定最大域宽则是比较危险的。因为输入太长时会导致超越参数给定的缓冲区，而无论给定的缓冲区有多长，用户仍可以给出比它更长的输入。

(5) 如下调用：

```
double da1, da2;
(void) scanf("%f %lf %", &da1, &da2);
printf("da1=%12.10e, da2=%12.101e",da1, da2);
```

对于输入 567890123456 567890123456 将产生输出：

```
da1=8.238678167e+91, da2=5.6789012346e+11
```

结果显示读入至变量 dal 中的值不对，它的值本应当与 da2 的值相同。错误原因是由于 da1 对应的转换区分符没有指明参数类型是 double *，即没有指定 l 标志。当参数类型长度与转换的默认类型长度不一致时，一定要注意指明长度标志。

对于浮点输入转换，默认的参数类型是 float *，这不同于对应的输出转换。对于输出转换，默认类型是 double。printf 的浮点参数是由默认参数类型提升而被转换为 double 的。但 float *不会被提升为 double，因为默认参数类型提升不会提升被指对象。

格式输入函数不像格式输出函数使用那么频繁，一个原因是其用法十分不灵活，正确地使用它匹配输入需要特别仔细小心，稍不小心便可能出现错误；另一个原因是它难于从匹配错误中恢复。

例 4-12 下面的程序利用 sprintf 函数把二进制数据转换为十进制字符串的形式。

```
1    /* ex12.c */
2    #include <stdio.h>
3    #include <math.h>
4    int main(int argc, char* argv[])
5    {
6         char buf[80];
7         float radius, area;
8         float pi=3.1419926;
9         if(argc!=2)
10        {
11             printf("Usage: %s radius \n",argv[0]);
12             return 1;
13        }
14        sscanf(argv[1],"%f",&radius);          /*从命令行读入半径 */
15        area=pi*radius*radius;
16        sprintf(buf,"The area of a circle with radius %f is %f\n", radius, area);
17        puts(buf);
18        return 0;
19   }
```

上述程序根据读入的圆半径计算圆的面积，计算结果用 sprintf 函数进行格式转换，以字符形式存储在数组 buf 中，并输出该结果。程序运行结果如下：

```
$ ./ex12 33              /*执行程序 */
The area of a circle with radius 33.000000 is 3421.629883
```

4.9 临 时 文 件

临时文件给应用程序提供了一种以文件形式暂存数据的手段，它可以存放计算的中间结果，也可以在关键操作之前用于备份数据。例如，编译程序在两遍扫描中间可以用临时文件存放第一遍的扫描结果；数据库应用程序在删除一个记录时可以用临时文件存放需要保存的记录，当进程结束时再将此临时文件更改为数据库中的正式文件，然后删除原来的文件。

临时文件的这种用途有一个先决条件：这就是文件名必须是唯一的；否则，可能由于其他进程使用相同的文件名而引起冲突。

唯一的文件名可以通过函数 tmpnam 和 tempnam 而获得：

```
#include <stdio.h>
char *tmpnam(char *s);
char *tempnam(const char *dir, const char *pfx);
```

tmpnam 返回一个与系统中已经存在的文件名不相同的临时文件名。如果参数 s 不是空指针，它也存储该文件名于 s 指出的字符串中。在参数 s 为空指针的情况下，tmpnam 返回值指向的字符串是静态分配的，下一次调用 tmpnam 会覆盖前一次的内容，因此，如果需要多次调用它的话，应当使用非空指针的 s 参数。s 参数指向的字符串大小应当至少不小于 L_tmpnam。在同一个进程内，tmpnam 至多可以调用 TMP_MAX 次，并且每一次调用生成的临时文件名各不相同。

tempnam 的功能与 tmpnam 相同，不同的是它可以指定临时文件存放的目录以及文件名的前缀。dir 参数给出目录的路径，pfx 参数给出文件名前缀。

tempnam 依次测试下述条件来确定目录路径名：

(1) 如果定义了环境变量 TMPDIR，用它的位作为目录；即它可以覆盖 dir 参数。

(2) 如果 dir 参数指向一个合法的目录字符串，用它作为目录。

(3) 如果 dir 参数是空指针或者指向一个非法目录路径名，使用<stdio.h>中宏变量 P_tmpdir 定义的目录；这个宏变量定义默认的临时文件目录。

(4) 如果 P_tmpdir 定义的目录不可访问，使用事先定义的目录，通常是/tmp。

另外，有许多应用程序喜欢以某种打头字符序列来命名临时文件名，这就需要使用 pfx 参数。这个参数可以是空指针，也可以指向 1~5 个字符的字符串。

tempnam 函数用 malloc 为所返回的文件路径名分配存储空间。由于这片空间是动态分配的，因此该函数是可重入的。不过，当不再需要它时，调用进程应当用 free 来释放它。

例 4-13 以下程序说明了 tmpnam 和 tempnam 函数的功能。

```
1      /* ex13.c */
2      #include <stdio.h>
```

```
3
4    int main(void)
5    {
6            char tmpname[L_tmpnam];
7            char *filename;
8            FILE *fp;
9            char *pfx="xxx";
10           printf("Temporary file name1 is %s\n",tmpnam(NULL));
11           tmpnam(tmpname);
12           printf("Temporary file name2 is %s\n",tmpname);
13           if((fp=fopen(tmpname,"w"))==NULL)
14           {
15                   printf("fopen failed\n");
16                   return 1;
17           }
18           else
19                   printf("write tempfile %s ok \n", tmpname);
20           printf("%s \n", tempnam("/tmp/lxy",pfx));
21           printf("%s \n",tempnam(NULL," "));
22           printf("%s \n",tempnam("tmp/lxy",pfx));
23           return 0;
24   }
```

执行这个程序的结果为：

```
$ ./ex13
Temporary file name1 is /tmp/fileqolcSo
Temporary file name2 is /tmp/filer0Wgmx
write tempfile /tmp/filer0Wgmx ok
/tmp/lxy/xxx5IQnQF
/tmp/ tDkvkO
/tmp/xxxoInDOW
```

如程序 13 所示，这两个函数仅仅生成一个唯一的临时文件名，它们并不实际创建临时文件，创建和删除临时文件都是应用程序自己的事情。但是，在文件被实际创建之前的这段时间内，其他进程有可能创建相同名字的文件，因为这两个函数是根据系统已经存在的文件来确定新名字的。

函数 tmpfile 能够避免这种竞争条件的出现，它在命名一个临时文件的同时也打开它。

```
#include <stdio.h>
FILE *tmpfile(void);
```

tmpfile 返回一个指向唯一临时文件的流指针，此临时文件以读 / 写方式(w+)被打开，并且当关闭了所有对它的引用时，它将被自动删除。

如果调用出错，tmpfile 返回 NULL。

例 4-14　下面的程序说明了 tmpfile 的用法。

```
1    /* ex14.c */
2    #include <stdio.h>
3    int main()
4    {
5        FILE *tempfp;
6        char line[256];
7        tempfp=tmpfile();
8        if(tempfp==NULL)
9        {
10           printf("tmpfile error!\n");
11           return 1;
12       }
13       printf("Opened a temporary file OK!\n");
14       fputs("One line of output \n",tempfp);
15       rewind(tempfp);
16       if(fgets(line, sizeof(line),tempfp)==NULL)
17       {
18           printf("fgets error!\n");
19           return 2;
20       }
21       fputs(line, stdout);
22       return 0;
23   }
```

上述程序首先利用 tmpfile 函数创建并打开一个临时文件(第 6 行)，然后往临时文件写入一行数据(第 15 行)，随后将它读出并写至标准输出(第 20 行)。

程序运行结果如下：

```
$ ./ex14
Opened a temporary file OK!
One line of output
```

4.10　小　　结

本章详细地介绍了标准输入输出的有关概念和操作。这些函数都是基于标准输入输出库的，标准 C 中都实现了这些操作，所以在这章中学到的东西也可以应用于其他的系统中。

读者学习完这一章之后，应该能毫无困难地编写涉及输入输出的程序了。

习 题

一、填空题

1. 当打开一个流时，标准输入输出函数返回一个______。

2. 有三个流是在执行程序时自动打开的。它们是______、______和______。

3. 有 3 种类型的无格式 I/O 函数可用来读写流，它们是______、______和______。

4. 每一个流对象内部都保持着两个指示器：一个为______，当读写文件出错时该指示器被设置；另一个为______，当遇到文件尾时该指示器被设置。

5. 流有 3 种不同的缓冲类型，它们是______、______和______。

二、选择题

1. 下面的函数不能用于打开流的是______。

   (A) fopen (B) freopen (C) fdopen (D) ffopen

2. 下列个函数不能一次读入一个字符的是______。

   (A) fgetc (B) fgetchar (C) getchar (D) getc

3. 块 I/O 有时也称为______。

   (A) 字符 I/O (B) 行 I/O (C) 列 I/O (D) 二进制 I/O

4. ______函数能够对输入输出数据进行诸如数据类型、精度、位置等格式控制。

   (A) 格式化 I/O (B) 字符 I/O (C) 行 I/O (D) 块 I/O

5. 下列函数不属于格式输出的是______。

   (A) printf (B) fprintf (C) scanf (D) sprintf

三、上机题

1. 编写一个程序，从键盘输入一个字符，并将其显示出来，当输入 q 时，程序退出。

2. 编写一个程序，打开一个文本文件，读入一行内容，然后在终端显示此行内容，其中文件名作为命令行参数。

3. 编写一个程序，打开一个文本文件，读入一行内容，然后把此行内容中的小写字母转换为大写字母，其他字符不变。其中文件名作为命令行参数。

4. 编写一个程序，从键盘中输入字符，并将它写入一个临时文件，当输入 q 时，程序退出。

CHAPTER 5

进 程 操 作

有关文件操作的讨论已告一段落，从本章起将开始研究 Linux 的多任务特征——进程。本章主要探讨了与进程基本控制有关的系统函数的使用，即怎样创建一个新进程，执行进程、终止进程以及进程等待等，接下来的一章是本章的延续，主要讨论有关进程通信方面的系统函数。

5.1 进 程 概 述

进程现在已成为操作系统和并发程序设计中的一个非常重要的概念。本节先介绍进程的基本概念，然后介绍了 Linux 系统中的进程。另外，由于 Linux 中进程具有多种识别号，我们专门用一小节介绍这方面的知识。

5.1.1 进程的基本概念

进程的概念起源于上世纪 60 年代，但到底什么是进程目前尚无统一、确切的定义。一般认为程序是存储在磁盘上包含可执行机器指令和数据的静态实体，而进程是具有一定功能的程序关于一个数据集合的一次运行活动，是处于活动状态的计算机程序。

进程在其生存期内可能处于三种基本状态：

- 运行态：进程占有 CPU，正在运行。
- 就绪态：进程本身具备运行条件，等待 CPU。
- 等待态：等待除 CPU 之外的其他资源或条件，不能运行。

进程在这几种状态之间相互转化，但对于用户是透明的。

进程是一个随执行过程不断变化的实体。和程序要包含指令和数据一样，进程也包含程序计数器和所有 CPU 寄存器的值，同时它的堆栈中存储着如子程序参数、返回地址以及变量之类的临时数据。当前的执行程序或者说进程，包含着当前处理器中的活动状态。在多处理操作系统中，进程具有独立的权限与职责。如果系统中某个进程崩溃，不会影响到其余的进程。每个进程运行在各自的虚拟地址空间中，通过一定的通信机制，它们之间才能发生联系。

5.1.2 Linux 进程

为了让 Linux 来管理系统中的进程，每个进程用一个 task_struct 数据结构来表示(task 即任务，它与进程在 Linux 中可以混用)。数组 task 包含指向系统中所有 task_struct 结构的指针。

这意味着系统中的最大进程数目受 task 数组大小的限制，缺省值一般为 512。创建新进程时，Linux 将从系统内存中分配一个 task_struct 结构并将其加入 task 数组。当前运行进程的结构用 current 指针来指示。

task_struct 数据结构庞大而复杂，但它可以分成一些功能组成部分：

状态(state)：除了上述三种进程的基本状态之外，Linux 进程还有 stopped 和 zombie 状态，在后续章节再做介绍；

调度信息：系统根据这些信息判定哪个进程最迫切需要运行；

进程标志号(Identifiers)：用来区分进程的标识。

进程间通信机制：Linux 支持经典的 Unix IPC 机制，如信号、管道和信号灯以及 System V 中机制，包括信号量、消息队列和共享内存。

5.1.3 进程的识别号(ID)

在 Linux 系统中，每一个进程都有唯一的进程识别号(process ID)，系统就根据这些进程识别号来管理进程。除此之外，每个进程还有一个真实用户识别号(real user ID)、一个真实组识别号(real group ID)、一个有效用户识别号(effective user ID)和一个有效组识别号(effective group ID)。一般情况下，真实用户 ID 与有效用户 ID 是相同的，都是运行该进程的用户 ID；当我们设置了 set_user_ID 标记位时，真实用户 ID 仍为运行进程的用户 ID，但有效用户 ID 变成当前运行文件的所有者的 ID。组识别号也是这样，区别就是相应的标记位为 set_group_ID。

这些听起来有些复杂，下面用一个简单的例子来说明这个问题。现在用户 A 运行程序 X 产生一进程 P。用户 A 的用户识别号是 10，组识别号是 1，程序 X 的所有者用户识别号为 100，组识别号是 2，当前 set_user_ID 位为 1，set_group_ID 为 0。那么进程 P 的真实用户识别号为 10，有效用户识别号为 100，真实组识别号与有效组识别号都为 1。有效用户

/组识别号常用来判断读写文件的优先权限。

可以调用下列函数取得进程的各种识别号：

```
uid_t getuid();        /* 返回真实用户识别号 */
uid_t getpid();        /*返回真实组识别号*/
uid_t geteuid();       /*返回有效用户识别号*/
uid_t getepid();       /*返回有效组识别号*/
pid_t getppid();       /*返回父进程识别号*/
pid_t getpgrp();       /*返回进程组识别号*/
```

这些函数在后面会进一步讲述。

5.1.4　进程调度

在 Linux 系统中，进程有两种运行模式：用户模式和系统模式。用户模式的权限比系统模式下的小很多，对于一般的进程，都是部分时间运行于用户模式，部分时间运行于系统模式。进程通过系统调用在这两种模式之间切换；当系统调用发生时，进程将由用户模式切换到系统模式继续执行；当系统调用返回时，进程将由系统模式切换回用户模式。

在 Linux 系统中，进程不能被抢占。只要能够运行它们就不会被停止。当进程必须等待某个系统事件时，它才决定释放出 CPU。进程常因为执行系统调用需要等待。由于处于等待状态的进程还可能占用 CPU 时间，所以 Linux 采用了预加载调度策略。在此策略中，每个进程只允许运行很短的时间(200ms)，当这个时间用完之后，系统将选择另一个进程来运行，原来的进程必须等待一段时间以继续运行。这段时间称为时间片。

可运行进程是一个只等待 CPU 资源的进程。Linux 使用基于优先级的简单调度算法来选择下一个运行进程。当选定新进程后，系统必须将当前进程的状态、处理器中的寄存器以及上下文状态保存到 task_struct 结构中。同时它将重新设置新进程的状态并将系统控制权交给此进程。为了将 CPU 时间合理地分配给系统中每个可执行进程，调度管理器必须将这些时间信息也保存在 task_struct 中。

在 task_struct 结构中保存的调度信息如表 5-1 所示。

表 5-1　进程调度信息

字 段 名	含　义
policy	该字段表示了进程的调度策略。系统中有两类进程——普通与实时进程。实时进程的优先级要高于普通进程。实时进程也有两种策略——时间片轮转和先进先出
priority	该字段表示了实时进程的相对优先级
rt_priority	该字段表示了实时进程的相对优先级
counter	该字段表示了进程允许运行的时间。进程首次运行时为进程优先级的数值，它随时间变化递减

5.2 进 程 控 制

通过前面一节介绍，读者已经对进程的概念有了一定的了解，本节主要介绍 Linux 下基于进程的系统调用，包括创建进程，使进程等待，结束进程，改变进程执行映像，改变进程执行的优先级等。这里介绍的都是进程的基本操作，为下面继续介绍信号的使用和进程间通信做一定的准备。

5.2.1 进程的创建

创建一个新进程的唯一方法是由某个已存在的进程调用 fork 或 vfork 函数，被创建的新进程称为子进程，已存在的进程称为父进程。当然，某些特殊进程并不需要通过这种方法来创建，如 swapper 进程、init 进程等，它们是作为系统启动的一部分而被内核创建的。本节将详细介绍 fork 与 vfork 两个函数的内在操作、使用方法及它们的异同点。

5.2.1.1 fork 函数

fork 函数是用来创建子进程的函数，具体调用如下所示。

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

fork 函数没有参数，它是一个单调用双返回的函数。具体地说，某个进程调用此函数后，若创建子进程成功，则这个函数在父进程中的返回值是创建的子进程的进程标识号，而在子进程中的返回值为零，否则(创建不成功)返回-1。每个进程可以有许多子进程，但没有哪个函数调用是用来将子进程的进程标识号返回给父进程的；而每个进程至多有一个父进程，利用 getpid 函数可以得到父进程的进程标识号。因此，fork 函数将子进程的进程标识号返回给了父进程，使得父进程利用此进程标识号与子进程取得联系；而统一给子进程返回零。这样可以很简单地区分父进程和子进程，这一点可以从例 5-1 中看出。

子进程是父进程的一个副本。具体说，子进程从父进程那里得到了数据段和堆栈段的副本，这些需要分配新的内存，而不是与父进程共享内存。对于只读的代码段，一般情况下，是使用共享内存的方式访问的。fork 函数返回后，子进程和父进程一样，都是从调用 fork 函数的语句的下一条开始执行。

因为创建子进程之后常常伴随着 exec 的调用，现行的许多实现机制实际上并不采用上述方法，将父进程的所有数据、堆、栈全都复制下来，而是使用一种“写时复制(COW)”的技术。父进程和子进程共享数据和堆栈区域，但是内核将它们都设置为只读权限。当任一进程要修改这些区域时，再生成该块内存的副本，通常是虚存的一页。

例 5-1　该实例程序说明 fork 系统函数执行成功后，生成的子进程从程序中间开始执行程序，此外还将说明父子进程的返回值问题，程序如下：

```
1	/* ex1.c */
2	#include <stdio.h>
3	#include <sys/types.h>
4	#include <unistd.h>
5
6	int main()
7	{
8	pid_t pid;
9	printf("Start of fork testing. \n");
10	pid=fork();
11	printf("Return of fork success:pid=%d\n",pid);
12	return 0;
13	}
```

编译后的程序命名为 ex1，运行该程序，其结果如下：

```
$ ./ex1
Start of fork testing.
Return of fork success:pid=0
Return of fork success:pid=7717
```

该程序首先输出了一行信息(第 9 行)，然后创建一个子进程(第 10 行)。当子进程成功建立后，子进程为就绪状态。当父、子进程都从 fork 系统函数返回时，处理机可能先调度到子进程。由于子进程已继承了父进程的执行环境，因此它也继承了父进程的执行地址，即从第 11 条语句开始执行程序，输出“Return of fork success：pid=0”后结束。然后处理机调度到父进程又输出了“Return of fork success pid=7717”，这是为什么该条语句执行两次的原因。从中可以看到，子进程从 fork 得到的返回值为 0，而父进程得到的返回值为 7717(7717 是子进程的进程标识号)，父、子进程得到了不同的返回值。

从例 5-1 中还可以看到子进程继承了父进程的正文段内容。

例 5-2　该程序说明子进程对父进程数据段的继承性，程序如下：

```
1	/* ex2.c */
2	#include <stdio.h>
3	#include <sys/types.h>
4	#include <unistd.h>
5	#include <sys/ipc.h>
6	#include <sys/shm.h>
7	#define SIZE 1024
8	#define KEY 99
9
```

```
10  int shmid;
11  int j=5;
12
13  int main()
14  {
15  int i, *pint;
16  pid_t pid;
17  char *addr;
18
19  i=10;
20  shmid=shmget(SIZE,KEY,IPC_CREAT|0777);
21  pint=shmat(shmid,0,0);
22  *pint=100;
23  printf("Start of fork testing \n");
24  pid=fork();
25  i++;j++;*pint+=1;
26  printf("Return of fork success:pi=%d\n",pid);
27  printf("i=%d, j=%d\n",i,j);
28  printf("*pint=%d\n",*pint);
29
30  return 0;
31  }
```

变量 i 是一个局部变量(第 15 行)，变量 j 是一个全局变量(第 11 行)，变量 pint 所指向的存储区是共享的(第 15 行)。下面看一看 fork 系统调用成功后父、子进程对这些变量操作的结果。程序第 20 行和第 21 行中的 shmget、shmat 是与共享内存段有关的系统函数。

该程序首先使用系统函数 shmget 申请了一共享内存段(第 20 行)，然后把该共享内存段用系统函数 shmat 附在进程的虚拟地址空间上(第 21 行)，该共享内存段的虚拟首地址存放在 pint 变量中。将该地址的内容赋成 100(第 22 行)。然后开始测试操作，先输出测试信息(第 23 行)，再调用 fork 创建一子进程(第 24 行)。剩下的语句为父子进程都将执行的代码段，先分别对局部变量 i、全局变量 j、共享量*pint 进行操作(第 25 行)，再输出一些信息看看这些变量的值(第 26、27 行)。

程序运行结果如下：

```
$ ./ex2
Start of fork testing
Return of fork success:pi=0
i=11, j=6
*pint=101
Return of fork success:pi=8003
i=11, j=6
*pint=102
```

局部变量 i 和全局变量 j 的值没有受到进程的影响，这是因为它们只在自己的进程空间里操作，互不干涉。而 pint 指向存储单元中的值是两个进程所共享的。从程序的运行结果看，该值与进程被调度到执行的时机有关，从本例的结果看：子进程先执行到对共享内存段进行操作的“*pint+=1;”语句，将 pint 指向的共享内存单元内容加 1，使其值变为 101。当子进程执行完后，父进程又执行到该语句，因子进程已将 pint 指向的内容改变为 101 了，父进程再操作后其内容变为 102。

从该程序的运行结果可以看到：如果父进程在创建子进程前申请了共享内存段，则子进程将同父进程一样有对该共享内存段进行操作的权力。由于处理机调度的随机性，共享内存段的内容随父子进程的随机调度可能变得不一致，因此在有共享内存段的程序中，使用 fork 系统调用一定要小心，进程要控制对共享内存段的同步操作。

以上两个实例说明了 fork 系统调用后，子进程和父进程正文段和数据段的继承性及其相互关系，下面的例 5-3 说明 fork 系统调用后父子进程间共享文件指针的问题。

例 5-3 该程序说明 fork 系统调用后父子进程间共享文件指针的问题，实现的功能为复制文件。程序如下：

```
1     /* ex3.c */
2     #include <sys/types.h>
3     #include <unistd.h>
4     #include <fcntl.h>
5
6     int rfd, wfd;
7     char c;
8
9     int main(int argc, char*argv[])
10    {
11    if(argc!=3)
12    {
13            printf("Usage %s sourcesfiel destfile. \n",argv[0]);
14            return 1;
15     }
16    if((rfd=open(argv[1], O_RDONLY))==-1)
17    {
18            printf("openf file %s failed.\n",argv[1]);
19            return 2;
20    }
21    if((wfd=creat(argv[2],0666))==-1)
22    {
23            printf("create file %s failed.\n",argv[2]);
24            return 3;
25    }
26    fork();
```

```
27  for(;;)
28  {
29        if(read(rfd,&c,1)!=1)
30        return 4;
31        write(wfd,&c,1);
32  }
33  return 0;
34  }
```

命令行中第二个参数指出的文件是源文件，它以只读方式打开(第 16 行)，第三个参数指出的文件为目标文件，它以读／写方式创建(第 21 行)。fork 系统函数创建一子进程成功后(第 26 行)，子进程继承父进程的 user 区，因此子进程同父进程共享父进程的打开文件，并具有和父进程相同的文件存取权限。此后父、子进程各自独立地执行 for 循环中的语句(第 27 行)，从源文件中读一字符(第 29 行)，将其写入到目标文件中(第 31 行)，直到读到文件尾为止。以下是该程序的一个执行结果：

```
$ ./ex3 srcfile dstfile          /*执行程序*/
lxy@lxy-desktop:~/test1/chapter6$ cat srcfile          /*显示源文件内容*/
How are you?
Fine.Thank you. And you?
I'm fine,too.
lxy@lxy-desktop:~/test1/chapter6$ cat dstfile          /*显示目标文件内容*/
How are you?
Fine.Thank you. And you?
I'm fine,too.
```

以上三个实例都是假设 fork 创建新进程总可获得成功，其实在某些情况下它也会执行失败。例如：系统中进程总数超过了规定的值或新进程所需求的空间不能得到满足时，fork 都将失败。

5.2.1.2 vfork 函数

vfork 函数的作用基本类似于 fork 函数，调用流程和返回值与 fork 函数完全相同，但是它们的语意很不相同。

用 vfork 创建的新进程的主要目的在于用 exec 函数执行另外的程序。这样，用 vfork 创建的新进程并不完全复制父进程的数据区，因为子进程只是为了执行另外的程序，而一般情况下，fork 调用之后马上就调用 exec，用不到父进程的数据段。实际上，在它还没有调用 exec 或 exit 之前，子进程的运行中，是与父进程共享数据段的。这一机制在 Linux 内存的虚拟页式管理的配合下，提供了很高的效率。

vfork 与 fork 的另一不同点表现在父子进程的执行次序上。fork 不对父子进程的执行次序进行任何限制，而 vfork 调用中，子进程先运行，父进程挂起。直至子进程调用 exec 或 exit，这之后，父子进程的执行次序不再有限制。这样，如果子进程在调用 exec 或 exit

之前需要父进程的进一步活动，就会造成死锁。

下面看一个简单的例子。

例 5-4　一个简单的 vfork 调用。

```
1     /* ex4.c */
2     #include <stdio.h>
3     #include <sys/types.h>
4     #include <unistd.h>
5
6     int global=4;
7
8     int main()
9     {
10            pid_t pid;
11            int vari=5;
12            if((pid=vfork())<0)
13            {
14                    printf("vfork error.\n");
15                    return 1;
16            }
17            else if(pid==0)
18            {
19                    global++;
20                    vari--;
21                    printf("Child changed the vari and global \n");
22                    _exit(0);
23            }
24            else
25                    printf("Parent didn't changed the vari and global\n");
26            printf("global =%d ,vari=%d \n",global,vari);
27            return 0;
28    }
```

利用 vfork 创建子进程后(第 12 行)，父进程挂起，子进程先运行，在父进程的数据段中修改了变量 global 的值(第 19 行)，执行输出“Child changed the vari and global”(第 21 行)，然后调用_exit(0)，子进程退出(第 22 行)。父进程继续运行，执行了两条输出语句(第 25、26 行)。读者可以看到变量 global 和 vari 确实是被子进程改变了，而且这个程序的执行结果是确定的，这都是由于 vfork 与 fork 的不同造成的。

读者可能注意到子进程的结束使用_exit(0)，在后面的章节中还要详细介绍。

程序的执行结果如下：

```
$ ./ex4
```

```
Child changed the vari and global
Parent didn't changed the vari and global
global =5 ,vari=4
```

5.2.2 exec 函数

fork 函数只是将父进程的环境复制到新进程中，而没有用新程序来初始化创建的子进程，因此它并不能执行一个新的目标程序，而这一点又是程序设计时所必须的，为此 Linux 系统提供了 exec 系统调用，以便用指定的目标程序更换进程的执行映象。系统中的绝大多数命令都是通过 exec 来执行的。不但 shell 进程所创建的子进程使用它来执行用户命令，shell 进程本身和它的祖先进程也是用 exec 来启动执行的。

exec 函数有 6 种不同的使用格式，但在内核中只对应一个入口。不同之处在于不同格式有不同的名字和调用参数。这 6 种调用格式是：

```
#include <unistd.h>
int execl(const char*pathname, const char *arg0,... /* (char *)0*/);
int execv(const char *pathname,char *const argv[]);
int execle(const char *pathname, const char *arg0,... /*(char *)0, char *const envp []*/);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /*(char *)0*/);
int execvp(const char *filename, char *const argv[]);
```

其中，参数 pathname 指出一个可执行目标文件的路径名；参数 filename 指出可执行目标文件的文件名；arg0 作为约定，同 pathname 一样指出目标文件的路径名；参数 argv 是一个字符指针数组，由它指出该目标程序使用的命令行参数表，按约定第一个字符指针指向与 pathname 或 filename 相同的字符串，最后一个指针指向一个空字符串，其余的指向该程序执行时所带的命令行参数；参数 envp 与 argv 一样也是一个字符指针数组，由它指出该目标程序执行时的进程环境，它也以一个空指针结束。

exec 调用成功后，调用进程的正文段被指定的目标文件的正文段所覆盖，其属性的变化方式与 fork 成功后从父进程那里继承属性的方式几乎是一样的，但以下几项是不同的：

(1) 进程标识号不变。

(2) 如设置了 close_on_exec 位，则不继承调用进程已打开的文件描述符，也就是在执行 exec 时关闭这些文件。

(3) 忽略调用进程先前捕获的信号。

(4) 忽略调用进程先前设置的文件创建屏蔽码。

调用成功时 exec 不返回，从而不执行 exec 以后的所有语句，失败时返回-1。

exec 的 6 种格式在以下三点上有所不同：

(1) pathname 是一个目标文件的完整路径名，而 filename 是目标文件名，它可以通过

环境变量 PATH 来搜索。

(2) 由 pathname 或 filename 指定的目标文件的命令行参数是完整的参数列表还是通过指针数组 argv 来给出的。

(3) 环境变量是系统自动传递还是通过 envp 传递的。

表 5-2 说明了 exec 函数的 6 种不同格式对以上三点的支持。

表 5-2　exec 的 6 种格式

函　　数	参 数 形 式	环 境 传 送	路 径 搜 索
execl	全部列表	自动	不
execv	指针数组	自动	不
execle	全部列表	不自动	不
execve	指针数组	不自动	不
execlp	全部列表	自动	是
execvp	指针数组	自动	是

下面看一个简单的例子：

例 5-5　该实例是一个非常简单的程序，只是为了说明 exec 的使用，程序如下：

```
1     /* ex5.c */
2     #include <stdio.h>
3     #include <unistd.h>
4
5     int main()
6     {
7             printf("===system call execl testing ===\n");
8             execl("/bin/date","/bin/date",0);
9             printf("exec error !\n");
10    }
```

程序的第 7、第 9 行输出了判断信息。当 execl 调用成功时(第 8 行)，因不执行旧程序中的语句，使第 9 行语句无法输出，该信息只当 execl 调用失败时，才能输出。运行该程序的一个结果如下：

```
$ ./ex5
===system call execl testing ===
2008 年 03 月 20 日 星期四 14:26:10 CST
```

如果将第 8 行的 execl 写成 execl("date","date",0)，则会有如下的运行结果：

```
$ ./ex5
===system call execl testing ===
```

```
exec error !
```

exec 调用失败，这是因为 execl 调用的第一个参数要求是一个目标文件的完整路径名，而不能只使用文件名。

如果将第 8 行的 execl 写成 execlp("date","date",0)，则会有如下的运行结果：

```
$ ./ex5
===system call execl testing ===
2008 年 03 月 20 日 星期四 14:39:54 CST
```

exec 调用成功，由此可以看出 execl 和 execlp 这两个函数可以替换使用。但需注意 execlp 中指定的目标文件名必须在 PATH 所定义的某个路径名之下。

例 5-6 该实例说明进程在 exec 调用成功后保持了调用进程已打开的文件描述符。程序如下：

```
1        /* ex6.c*/
2      #include <stdio.h>
3        #include <sys/types.h>
4        #include <sys/stat.h>
5        #include <fcntl.h>
6
7        int main()
8        {
9                int fd;
10               char buf[6];
11               fd=open("./test.c",0);
12               read(fd,buf,6);
13               printf("====output in main()= ===\n%s\n",buf);
14               execl("./test","./test",0);
15               printf("exec error \n");
16       }
------------------------------------------------------------------------------------------------
1        /* test.c */
2        #include <stdio.h>
3        #include <sys/types.h>
4        #include <sys/stat.h>
5        #include <fcntl.h>
6
7        int main()
8        {
9                int i,fd;
10               char buf[1000];
11               fd=open("./ex6.c",0);
```

```
12                printf("====fd=%d in test.c ====\n",fd);
13                fd-=1;
14                read(fd,buf,1000);
15                printf("=== output in test.c === %s \n",buf);
16          }
```

在 ex6.c 中先打开 test.c 文件(第 11 行)，并读出 6 个字符加以显示(第 12 行)，接着调用 execl 改换进程映像(第 14 行)，执行 test(由编译 test.c 而形成的目标文件)，test.c 中打开另一个文件 ex6.c(第 11 行)，获取的文件描述符减 1 后作为新的文件描述符(第 13 行)，再从该文件描述符所对应的文件中读取信息并显示(第 14、15 行)。

下面是该程序的执行结果，先单独执行 test，而后再执行 ex6：

```
$ ./test
====fd=3 in test.c ====
1234                              /* 程序停止在这里，锭入一些字符例如“1234”后继续执行
=== output in test.c === 1234     /* 显示的是“1234”*/
```

读操作时的文件描述符为 2，它是一个标准的文件描述符，代表错误输出。接着执行 ex6：

```
$ ./ex6
====output in main()= ===
#inclu
====fd=4 in test.c ====
=== output in test.c === de <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
        int i,fd;
        char buf[1000];
        fd=open("./ex6.c",0);
        printf("====fd=%d in test.c ====\n",fd);
        fd-=1;
        read(fd,buf,1000);
        printf("=== output in test.c === %s \n",buf);
}
```

从中可以看出：调用 execl 执行 test 程序时，它打开 ex6.c 所得到的文件描述符为 4 而不是 3，此时文件描述符为 3 的文件是 test．c，所以读取的文件为 test.c。test 执行时继承了 execl 调用前的所有打开文件描述符以及相应的文件读／写指针，因此 test 程序所显示

的内容是除去先前读取的 6 个字节后的 test.c 文件的内容。

如果在 ex6.c 的 execl 系统函数前增加以下程序行 fcntl(fd,F_SETFD,1);即设置执行时关闭标志 close_on_exec，则在同一执行环境下，再执行 ex6 程序，结果为：

```
$ ./ex6
====output in main()= ===
#inclu
====fd=3 in test.c ====
1234                          /*程序停止在这里，键入一些字符例如“1234”后继续执行  */
=== output in test.c === 1234 /*  显示的是“1234”  */
```

从中可以看出：execl 启动执行 test 后，test 打开 ex6.c 文件所得到的文件描述符为 3，和单独执行 test 程序的情形是一样的。这是因为 fcntl 系统调用为该文件描述符设置了 close_on_exec 标记，当随后使用 exec 系统调用时，如执行成功，将不继承那些已用 fcntl 系统函数设置了 close_on_exec 标记的文件描述符；如 exec 执行失败，则这些文件描述符仍是合法的。

例 5-7 该实例讨论 exec 系统函数和 fork 系统函数的联合使用。它是一个实用工具，功能是对某一指定文件进行监视，当所监视的文件修改后，自动为它建立一个副本。程序的实现如下：

```
1       /* ex7.c */
2       #include <stdio.h>
3       #include <sys/types.h>
4       #include <unistd.h>
5       #include <sys/stat.h>
6       #include <fcntl.h>
7       int main(int argc, char *argv[])
8
9       {
10          int fd;
11          int stat,pid;
12          struct stat stbuf;
13          time_t old_time=0;
14          if(argc!=3)
15          {
16              fprintf(stderr,"Usage: %s watchfile copyfile \n", argv[0]);
17              return 1;
18          }
19          if((fd=open(argv[1],O_RDONLY))==-1)
20          {
21              fprintf(stderr,"Watchfile: %s can't open \n", argv[1]);
22              return 2;
```

```
23            }
24            fstat(fd, &stbuf);
25            old_time=stbuf.st_mtime;
26            for(;;)
27            {
28                fstat(fd, &stbuf);
29                if(old_time!=stbuf.st_mtime)
30                {
31                    while((pid=fork())==-1);
32                    if(pid==0)
33                    {
34                        execl("/bin/cp","/bin/cp",argv[1],argv[2],0);
35                        return 3;
36                    }
37                    wait(&stat);
38                    old_time=stbuf.st_mtime;
39                }
40                else
41                    sleep(30);
42            }
43        }
```

在命令行参数检查正确的情况下(第 14-18 行)，先打开要监视的文件(第 19-23 行)，使用 fstat 将需监视文件的状态取出(第 24 行)，然后把该文件的最后修改时间保留起来以备后用(第 25 行)；在这之后程序进入 for 循环(第 26 行)，for 循环中的代码完成以下工作——如果所监视的文件当前已被修改(第 29 行)，则使用 fork 系统函数创建一子进程(第 31 行)，子进程调用 exec 系统函数将监视文件复制到用户指定的文件中(第 32-36 行)；与此同时父进程调用另一个系统函数 wait 等待子进程终止(第 37 行)。当子进程终止后，父进程将被监视文件新的修改时间再保留起来以备下次使用(第 38 行)。如果监视文件没有被修改，则睡眠 30 秒(第 41 行)，然后接着做 for 循环中的工作，直到该进程被中断为止(第 31 行)。该程序以后台方式执行：

```
$ ./ex7 file file.bak&
```

此时，系统中将出现一个后台进程，此后如果 file 文件被修改，则该后台进程将创建子进程，由子进程 file 文件建立副本文件 file.bak。

5.2.3　结束进程

在 Linux 中，有三种正常结束进程的方法，两种异常终止的方法，列举如下。

(1) 正常结束

- 在 main 函数中调用 return。这相当于调用 exit。
- 调用 exit 函数。按这个函数在 ANSI C 中的定义，调用时将执行所有注册过的 exit 句柄，关闭所有的标准 I/O 流，但是并不处理文件描述符、多进程(父进程与子进程)、作业等，因而对 Linux 系统而言并不完善。
- 调用_exit 函数。_exit 被 exit 调用，关闭一些 Linux 特有的退出句柄。

(2) 异常终止

- 调用 abort。这其实是第二种情形的特例，因为它产生一个 SIGABRT 信号。
- 进程收到特定信号(关于信号将在下一章中详细介绍)。这个信号可以是进程自己产生的(如调用 abort 函数)，也可以来自其他进程或内核。例如，进程企图访问越界的内存地址或是除数为零时，内核都会产生信号中断进程。

不管用何种方式结束进程，最终都要执行内核的同一段代码。这段代码关闭该进程打开的所有文件描述符，释放占用的内存等。

exit 和_exit 函数的具体调用形式如下：

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

参数 status 为退出的状态。

在任何一种方式下，我们都希望结束的子进程能通知父进程自己是如何结束的。调用 exit 和_exit 退出时，会有一个退出状态字(exit status)作为参数传递给函数。最终执行内核代码时，内核依据退出状态字生成终止状态字(termination status)。当异常终止时，内核(并非进程！)直接产生一个终止状态字，描述异常终止的原因。可以通过 wait 或者 waitPid 函数来获得终止状态字，后面会进一步介绍。

有时候一个子进程结束时，它的父进程恰好忙于其他事务，不能接收子进程的终止状态。如果此时子进程完全消失了，那么当父进程处理完其他事务后想要检查子进程情况时，就没有可用的信息了。为此，内核为每个已结束的进程保留一定的信息，一般至少包含进程 ID、终止状态字、进程 CPU 时间等信息，于是，任何时候父进程调用 wait 或者 waitpid 函数都能得到想要知道的东西。等到父进程提取了状态字，内核再将保存这些信息的数据结构释放。这种已经结束，但其父进程尚未检查其终止状态的进程，术语称之为 zombie。如果一个程序调用了很多子进程，又不用 wait 调用取得它们的终止状态字，系统中就会有很多 zombie。

另外一个问题就是父进程可能先于子进程结束。这时，init 进程就会自动成为该子进程的父进程。通常的实现机制是，当一个进程结束时，系统逐一检查所有的活动进程，如果某进程的父进程是这个被结束的进程，系统就将这个活动进程的父进程 ID 置为 1，即 init 的 ID 号。这样，就保证了每个进程都有它的父进程。

从上面的叙述中可以看出，我们经常引用的那个经典例子 helloword：

```
#include <stdio.h>
main()
{
 printf("hello world \n");
}
```

其实是有问题的。这个程序结束时什么也没做，也没有返回退出状态字，因而还不完整。写程序时千万别忘了在每个过程的最后加上一句 return 0 或 exit(0)。

5.2.4　进程等待

当一个进程结束时，会产生一个终止状态字，然后内核发出一个 SIGCHLD 信号通知父进程。因为子进程的结束对于父进程是异步的，因而这个 SIGCHLD 信号对于父进程也是异步的，父进程可以不响应，也可以调用 wait 或 waitpid 函数进行处理。默认情况是不响应。

wait 和 waitpid 函数格式如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

一个进程调用 wait 或 waitpid 函数，可能产生 3 种情况：

- 如果所有子进程都还在运行，进程挂起。
- 如果恰有子进程结束，它的终止状态字正等待父进程提取，立即得到该终止状态字并返回，其返回值为该子进程的进程号。
- 如果该进程没有子进程，立即返回，返回值为-1。

可以看出，如果在收到 SIGCHLD 信号后调用这两个函数，由于有子进程结束，因此可以立即正确返回；否则，在程序某一点随机地调用，则进程很可能被挂起。这时，如果进程有多个子进程，就等待直至有一个子进程结束，然后返回该子进程的终止状态字，父进程可以根据返回的进程 ID 判断是哪个子进程结束了。

参数 statloc 是一个整数的指针。如果它不为空，子进程的终止状态字就被存放在该参数指定的内存位置。如果我们不关心终止状态字，传入一个空指针就可以了。

传统上，返回的这个状态字是一个整数，特定位表示特定的信息，如某些位表示退出状态(对于正常结束的进程)，某些位表示终止原因(对于异常终止的进程)，还有其他一些标记位，具体实现可能略有差异。系统为程序员屏蔽了实现的细节，在 sys/wait.h 中定义了几个宏，只需调用这几个宏就可以得到所关心的信息。它们分别如表 5-3 所示。

表 5-3 wait.h 中定义的宏

宏	含 义
WIFEXITED(status)	当子进程正常结束时返回为真
WIFSIGNALED(status)	当子进程异常结束时返回为真
WEXITSTATUS(status)	当 WIFEXITED(status)为真时调用，返回状态字的低 8 位 bit 值
WTERMSIG(status)	当 WIFSIGNALED(status)为真时调用，返回引起终止的信号代号

下面来看一个具体的例子：

例 5-8 不同类型结束进程的状态字示例。

```
1	/* ex8.c*/
2	#include <sys/types.h>
3	#include <sys/wait.h>
4	#include <stdio.h>
5	/* 处理并打印状态字的子函数*/
6	void h_exit(int status)
7	{
8		if(WIFEXITED(status))
9			printf("normal termination, exit status=%d \n", WEXITSTATUS(status));
10		else if(WIFSIGNALED(status))
11			printf("abnormal termination, signal number =%d %s\n", WTERMSIG(status),
12		#ifdef WCOREDUMP
13			WCOREDUMP(status) ? "(core file generated)" : " )");
14		#else
15			") ");
16		#endif
17	}
18	 /*主函数。示范三种结束进程的不同方式，并调用 h_exit 函数处理返回状态字*/
19	int main()
20	{
21		pid_t pid;
22		int status;
23		/*子程序正常退出 */
24		if((pid=fork())<0)
25		{
26			printf("fork error \n");
27			exit(0);
28		}
29		else if (pid==0)
30			exit(7);
31		if(wait(&status)!=pid)		/*等待子进程*/
```

```
32          {
33              printf("wait error \n");
34              exit(0);
35          }
36          h_exit(status);             /*打印状态 */
37          /*子进程 abort 终止 */
38          if((pid=fork())<0)
39          {
40              printf("fork erro\n");
41              exit(0);
42          }
43          else if(pid==0)             /*子进程*/
44              abort();                /*产生信号 SIGABRT 终止进程*/
45          if(wait(&status)!=pid)      /*等待子进程*/
46          {
47              printf("wait error.\n");
48              exit(0);
49          }
50          h_exit(status);             /*打印状态*/
51          /* 子进程除零终止 */
52          if((pid=fork())<0)
53          {
54              printf("fork erro\n");
55              exit(0);
56          }
57          else if(pid==0)             /*子进程 */
58              status /=0;             /*除数为 0 产生 SIGFPE */
59          if(wait(&status)!=pid)
60          {
61              printf("wait error.\n");
62              exit(0);
63          }
64          h_exit(status);             /*打印状态*/
65
66          exit(0);
67      }
```

以上程序代码的第 6~17 行定义了 h_exit 函数，该函数用到了上面介绍的几个宏来处理和打印状态字。主函数 main 创建了三个子进程(第 24、38、52 行)，并且用不同的方式结束(第 30、44、58 行)，然后三次调用 h_exit 打印子进程的终止状态字(第 36、50、64 行)。在一些系统中 abort 系统函数会产生一个 core dump，导致整个程序结束；必须在程序中捕捉并处理 SIGABORT 信号，才能使程序继续执行——这就是那段看起来很奇怪的 ifdef 代

码的作用(第 12~16 行)。程序运行结果如下：

```
$ ./ex8
normal termination, exit status=7
abnormal termination, signal number =6 (core file generated)
abnormal termination, signal number =8 (core file generated)
```

前面已经介绍过，如果某进程有多个子进程，wait 返回任一子进程的终止状态字。那么如果我们要等待某个特定的子进程结束(假设我们知道要等待的子进程的 ID)，该怎么办呢?可以循环调用 wait 函数，然后将返回的进程 ID 与我们感兴趣的那个进程 ID 比较，如果两者不等就将有关信息保存起来，再继续等待，直至等到我们所等待的那个进程结束。下一次类似操作时，又要先遍历一遍以前保存过的已结束的子进程信息，看其中是否有我们感兴趣的进程，再根据情况调用 wait 函数。显然，这样循环十分麻烦。幸好 Linux 提供了另外一个函数 waitpid，使用这个函数不仅可以指定等待某个进程，而且还提供了其他一些参数选择，具有很大的灵活性。

waitpid 函数的具体调用形式在前面已经给出了，下面来看一看各参数的含义。参数 pid 的含义根据其取值而不尽相同，如表 5-4 所示。

表 5-4　参数 pid 的取值含义

值	含　义
<-1	等待进程组 ID 等于 pid 的绝对值的子进程
-1	等待任何子进程。这种情况下，相当于 wait 函数
0	等待进程组 ID 与父进程组 ID 相同的子进程
>0	等待进程 ID 等于 pid 的子进程

waitpid 函数返回子进程的进程 ID，子进程的终止状态字存放在 statloc 指向的地址中。调用 wait 函数只有一种情形可能产生错误，即调用进程无子进程，而调用 waitpid 还可能有其他错误情形，例如指定进程或进程组不存在，或者不是调用进程的子进程。

参数 options 进一步控制 waitpid 的运行。它可以取零，也可以取一些常数，或是常数的组合(用 OR 得到)。一个常用的常数是 WNOHANG，使用这个常数，即使 pid 指定的子进程的终止状态字不能立即得到，waitpid 也不会挂起，而是返回零。

事实上，wait 是 waitpid 的一个特例，就等于 waitpid(-1,*statloc,0)。下面看一个例子。

例 5-9　该例程说明 waitpid 的使用，程序实现如下：

```
1	/* ex9.c */
2	#include <sys/types.h>
3	#include <sys/wait.h>
4	#include <stdio.h>
5	int main()
6	{
```

```
7           pid_t pid;
8
9           if((pid=fork())<0)
10          {
11                printf("fork error.\n");
12                exit(0);
13          }
14          else if(pid==0)
15          {
16                if((pid=fork())<0)
17                {
18                      printf("fork error.\n");
19                      exit(0);
20                }
21                else if(pid>0)
22                      exit(0);
23                sleep(2);
24                printf("second child, parent pid=%d \n", getppid());
25                exit(0);
26          }
27
28          if(waitpid(pid, NULL, 0)!=pid)
29          {
30                printf("waitpid error.\n");
31                exit(0);
32          }
33          exit(0);
34    }
```

在上面这个程序中，进程创建了第一个子进程(第 9 行)，第一个子进程又创建了第二个子进程(第 16 行)，然后第一个子进程立刻结束(第 22 行)。第二个子进程是第一个子进程的子进程，而不是原进程的子进程，因此第一个子进程结束后它就成为 zombie 进程，显示的父进程 ID 为 1，即 init 进程的 ID 号(第 24 行)。主进程接收到第一个子进程的终止状态字就正常退出(第 28~33 行)，而第二个子进程因为先睡眠了 2 个周期(第 23 行)，在主进程结束之后才完成，因此运行结果如下(注意第二行行首的$符号，因为第二行输出之前主进程已经运行完了)：

```
$ ./ex9
$ second child, parent pid=1
```

5.2.5 system 函数

如果能在程序中使用命令行是十分方便的。例如，假设我们需要将日期和时间写入到某个文件中去，使用命令行形式，只需：

```
system("data>file");
```

即可。标准 C 语言中定义了 system 函数，但是它的操作紧密地依赖于操作系统。本节将详细介绍这个函数。

这个函数的调用形式如下：

```
#include <stdlib.h>
int system(const char *cmdstring);
```

这个函数是用 fork，exec，waitpid 三个系统函数实现的，返回值相对比较复杂。

(1) 如果 cmdstring 为空指针，当系统实现了 system 函数时，返回非零指针，否则返回零。这是个用来测试系统的 system 函数是否有效的方法。在一般的 Linux 系统中，system 函数都是有效的。

(2) 如果 cmdstring 不为空，就要根据 fork，exec，waitpid 三个系统函数的执行情况确定返回值。若 fork 出错或 waitpid 中出现非 EINTR 错误，system 返回-1。

(3) 如果 exec 错误返回，表示 shell 无法执行这个命令行。返回值与 shell 执行 exit(127)的返回值相同。

(4) 否则，若三个系统函数调用都成功了，返回值为 shell 的结束状态，与前面介绍的 waitpid 的返回值情况相同。

下面是 system 函数的实现代码，它可以帮助理解上面介绍的返回值情况。

```
1       #include <sys/types.h>
2       #include <sys/wait.h>
3       #include <errno.h>
4       #include <unistd.h>
5
6       int system(const char *cmdstring)
7       {
8         pid_t pid;
9         int status;
10        if(cmdstring==NULL)
11                return (1);
12        if((pid=fork())<0)
13                status=-1;
14        else if(pid==0)
15        {
```

```
16              execl("/bin/sh","sh","-c",cmdstring, (char *)0);
17              _exit(127);
18          }
19          else
20          {
21              while(waitpid(pid, &status,0)<0)
22                  if(errno!=EINTR)
23                  {
24                      status=-1;
25                      break;
26                  }
27          }
28          return(status);
29      }
```

分析：命令 sh 的“-c”选项指出(第 16 行)，下一个命令行参数就是命令输入，而不是从标准输入或文件中读取。sh 命令自动对这个以空字符结尾的字符串进行分析，将它分解为独立的命令行参数的形式，并执行这个命令。这个命令可以包括输入输出重定向、管道命令等。

如果不使用 sh 命令文件，而试着自己执行这个命令，那就困难多了。我们必须将这个命令串分解为独立的命令参数，然后调用 execlp 来执行，并且自己处理输入输出重定向、管道命令等。

注意子进程退出时使用了_exit 而不是 exit，这是为了保证子进程的标准 I/O 流不被清掉。

下面用一个小程序来测试一下这个函数调用。

例 5-10 一个简单的 system 函数调用程序。

```
1       /* ex10.c*/
2       #include <stdio.h>
3       #include <sys/types.h>
4       #include <sys/wait.h>
5       #include <stdlib.h>
6
7       int main()
8       {
9         int status;
10        if((status=system("date"))<0)
11        {
12            printf("system error .\n");
13            exit(0);
14        }
15        printf("exit status = %d \n",status);
16        if((status=system("nosuchcommand"))<0)
```

```
17          {
18                printf("system error");
19                exit(0);
20          }
21          printf("exit status =%d \n",status);
22          if((status=system("who; exit 44"))<0)
23          {
24                printf("system    error");
25                exit(0);
26          }
27          printf("exit status=%d\n",status);
28          exit(0);
29        }
```

将这个程序编译链接后，成为可执行文件 ex10。ex10 的执行情况如下：

```
$ ./ex10
2008 年 03 月 24 日 星期一 21:36:45 CST
exit status = 0
sh: nosuchcommand: not found
exit status =32512
lxy         tty7          2008-03-24 19:23 (:0)
lxy         pts/0         2008-03-24 19:25 (:0.0)
lxy         pts/1         2008-03-24 21:19 (:0.0)
exit status=11264
```

程序说明：第一个 system 调用，执行命令 date，正确输出当前系统时间(第 10 行)。而第二个 system 调用中，由于命令行为一个非法命令，系统无法执行，运行 execl("/bin/sh","sh","-c",cmdstring, (char *)0);时错误返回，接着运行_exit(127)指令，输出错误信息“sh：sh: nosuchcommand: not found”(第 16 行)。第三个 system 系统调用正确执行，返回值为 shell 的退出状态 11264(第 22 行)。

通过这个例子，应该更深入地理解了 system 函数。使用 system 函数调用来完成命令行的执行，而不直接用 fork 和 exec，这是因为 system 函数中进行了必须的错误处理和信号处理，这更有利于编程。

5.2.6　进程的用户标识号管理

每个进程的用户标识号有两个：实际用户标识号(real user id)和有效用户标识号(effective user id)，其对应的组标识号分别称为实际组标识号(real group id)和有效组标识号(effective group id)。一般而言，进程的实际用户标识号为运行该进程的用户标识号，它通常只用于系统记账。其他功能由有效用户标识号来完成，例如用有效用户标识号来检查对

文件的存取权限。一般情况下，一个进程的有效用户标识号与实际用户标识号是相等的，但系统允许改变进程的有效用户标识号。

系统提供了一组系统调用，用来管理进程的用户标识号，它们的格式是：

```
#include <sys/types.h>
#include <unistd.h>
unsigned short getuid();
unsigned short geteuid();
unsigned short getgid();
unsigned short getegid();
int setuid(uid_t uid);
int setgid(gid_t gid);
```

参数和功能说明：

前四个系统函数没有参数，分别返回调用进程的实际用户标识号、有效用户标识号、实际用户组标识号和有效组标识号。这些系统调用总能执行成功，不会发生任何错误。系统调用 setuid 和 setgid 用于设置进程的实际用户(组)标识号相有效用户(组)标识号。如调用进程的有效用户标识号是超级用户标识号，则将调用的进程实际用户(组)标识号和有效用户(组)标识号设置为 uid 或 gid，如调用进程的有效用户标识号不是超级用户标识号，但它的实际用户(组)标识号等于 uid 或 gid 时，则其有效用户(组)标识号被置成 uid 或 gid；如调用进程的实际用户(组)标识号不等于 uid 或 gid；并且它的有效用户标识号不是超级用户标识号，则 setuid 或 setgid 调用失败。调用 setuid 或 setgid 成功时，返回零，否则返回-1。

例 5-11 下面的程序说明了以上系统函数的使用，程序代码如下：

```
1        #include <sys/types.h>
2        #include <unistd.h>
3        #include <stdio.h>
4
5        int main(int argc, char *argv[])
6        {
7          int i,ret;
8          if(argc!=2)
9          {
10               printf("Usage %s num\n",argv[0]);
11               exit(1);
12         }
13         i=atoi(argv[1]);
14         printf("Before uid = %d ,euid = %d \n",getuid(),geteuid());
15         ret=setuid(i);
16         printf("After uid = %d ,euid =%d\n",getuid(),geteuid());
17         printf("ret=%d\n",ret);
```

```
18        }
```

分析：以上程序从命令行接受用户标识号(第 5 行)，用 getudi 和 geteuid 函数先取得进程的实际和有效用户标识号(第 14 行)，然后用 setuid 函数将进程的实际和有效用户标识号改为用户输入的数字(第 15 行)，随后再用 getuid 和 geteuid 函数取得改变后的实际和有效用户标识号。(第 16 行)，并输出 setuid 的返回值(第 17 行)。

下面分 2 种情况讨论该程序的执行：

(1) 如果执行该程序的用户为超级用户，只要命令行给出了用户标识号，无论所给的用户标识号是否存在，执行总能成功。

```
# ./ex11 111
Before uid = 0 ,euid = 0
After uid = 111 ,euid =111
ret=0
```

结果分析：将进程的实际和有效用户标识号都改为 111。

(2) 如果执行该程序的用户为一般用户，其用户标识号为 1000

```
$ ./ex11 111
Before uid = 1000 ,euid = 1000
After uid = 1000 ,euid =1000
ret=-1
```

结果分析：setuid 系统调用执行失败，原因是实际用户标识号 1000 不等于命令行参数中的 111。

```
$ ./ex11 1000
Before uid = 1000 ,euid = 1000
After uid = 1000 ,euid =1000
ret=0
```

结果分析；setuid 系统函数调用执行成功，原因是实际用户标识号 1000 等于命令行参数中的 1000。

注册程序 login 是一个典型的调用 setuid 系统函数的程序，login 进程的有效用户标识号是超级用户，该进程在建立用户的 shell 进程前，调用 setuid 将其实际和有效用户标识号调整为用户注册时的用户标识号。

5.2.7　进程标识号管理

系统中的每个进程都有唯一的非负整数作为其标识，它被称为进程标识号(pid)。系统使用进程标识号来管理当前系统中的进程。为了对具有某类特性的进程统一管理，系统又引入了进程组概念，并以组标识号来区别进程是否同组。一个进程的标识号是由系统为之

分配的，不能被修改；进程的组标识号是从父进程那里继承下来的，所以，通常情况下进程的组标识号是和它相关联的终端注册进程的进程标识号，组标识号可以通过 setpgrp 系统函数修改。

Linux 系统提供了一组系统调用用于管理进程标识号，它们的格式是：

```
#include <sys/types.h>
int getpid();
int getpgrp();
int getppid();
int setpgrp();
```

参数和功能说明：

前三个系统函数分别返回调用进程的进程标识号、进程组标识号和其父进程标识号。它们总能成功地返回。第四个调用设置进程组标识号，它将调用进程的进程组标识号改为调用进程的进程标识号，使其成为进程组首进程，并返回这一新的进程组标识号。

例 5-12　该实例说明 getpid 的使用——使用进程标识号作为临时文件的文件名，因为系统中当前存在进程的进程标识号是唯一的。程序代码为：

```
1        /*ex12.c */
2        #include <sys/types.h>
3        #include <stdio.h>
4
5        char file[20],string[8];
6        char *tmp="./tmp.";
7        int main()
8        {
9          char *s;
10         int fd;
11         s=(char *)malloc(10);
12         sprintf(s,"%s",getpid());
13         strcpy(file,tmp);
14         strcat(file,s);
15         if((fd=creat(file,0644))==-1)
16         {
17             fprintf(stderr,"file: %s create failed.\n",file);
18             exit(1);
19         }
20         write(fd,"TMP file",9);
21         close(fd);
22         exit(0);
23       }
```

分析：该程序利用 getpid 函数获取当前进程号(第 12 行)，在当前目录下生成一临时文件，其文件名以运行该程序的进程标识号为后缀，例如：tmp.xxx(第 15~20 行)。程序运行结果如下：

```
$./ex12
1112
$ cat tmp.1112
TMP file
```

从上面可看到，在当前目录下新产生了一个名为 tmp.1112 的文件，该文件含有程序中写入的内容：TMP file。

例 5-13 该实例说明系统调用 setpgrp 的使用，程序代码如下：

```
1       /* ex13.c */
2       #include <sys/types.h>
3       #include <stdio.h>
4       int main(int argc, char *argv[])
5       {
6         if(argc<=1)
7         {
8               fprintf(stderr, "Usage: %s command [arg ...]\n",argv[0]);
9               exit(1);
10        }
11        argv++;argc--;
12        if(fork()==0)
13        {
14              setpgrp();
15              execvp(*argv,argv);
16              printf("%s is not executed \n",*argv);
17              exit(0);
18        }
19        exit(0);
20      }
```

分析：该程序在检测命令行使用正确后(第 6~11 行)，创建一子进程(第 12 行)，子进程使用 setpgrp 系统函数改变子进程组标识号(第 14 行)，然后用 execvp 执行用户在命令行提交的命令(第 15 行)。父进程只完成子进程的创建工作，无论创建是否成功都不做任何其余工作，在 fork 后立刻消亡(第 19 行)。

该程序通过调用 setpgrp 来改变调用进程的进程组标识号，使进程不受注册进程的影响。当用户注销时，因注册进程已非该进程所在组的首进程，故该进程不接收注册进程发出的 SIGHUP 信号，从而不被终止，所以其功能类似于 nohup 命令。以下是该程序的执行结果：

```
$ ./ex13 du -s>./tmp
$ cat tmp
248
```

5.3　综合应用实例

这一节分析两个综合实例设计，第一个实例重点说明 fork 和 exec 系统函数，第二个实例重点说明 wait 和 exit 系统函数。通过实例的分析以期进一步说明本章所述系统函数之间的相互关系，以及使用这些函数进行编程的方法及应注意的事项。

例 5-14　该实例是一个交互式命令处理程序，它能完成 Linux 系统标准 shell 的小部分功能，具体是：

(1) 提交命令的参数最多为 8 个。

(2) 可前、后台执行。

(3) 一命令行中可同时拥有多个命令，彼此之间由分号隔开。

为方便说明，将其取名为 minish，实现 minish 程序的主流程如下：

```
1       for(;;)
2       {
3         output("mini_SH-->");
4         readcmd();
5         docommand();
6       }
```

该程序先输出命令提示符“mini_SH-->”(第 3 行)，等待用户键入命令，然后通过 readcmd 函数分析得到的命令行(第 4 行)，用 docommand 函数执行命令(第 5 行)。该命令结构是 Linux 系统中绝大部分交互式命令的实现模式。

mini_SH 的程序代码如下：

```
7       /*ex14.c*/
8       #include <sys/types.h>
9       #include <unistd.h>
10      #include <stdio.h>
11
12      #define MAXARG 10
13      #define LINSIZ 80
14      #define CMDSIZ 8
15
16      extern char **environ;
```

```
17      char *quit="quit.quit";
18      char cmdbuf[CMDSIZ][LINSIZ];
19      int cmdflag[CMDSIZ];
20
21      int main()
22      {
23        int i;
24        for( ; ;)
25        {
26              printf("mini_SH-->");
27              for(i=7;i>=0;i--)
28              {
29                      cmdflag[i]=0;                   /*清除后台标志 */
30                      cmdbuf[i][0]='\0';
31              }
32              if(i=readcmd())                         /*读命令行*/
33                      docommand(i);                   /*执行命令*/
34              else
35                      printf("read command failed, try again!!!\n");
36        }
37      }
38
39      readcmd()
40      {
41        char c,*p;
42        int i=0;
43        p=cmdbuf[0];
44        while((c=getchar())!='\n')
45        {
46              if(c==';')
47              {
48                      *p='\0';
49                      if(++i==6)
50                              return(++i);
51                      p=cmdbuf[i];
52              }
53              else if(c=='&')
54              {
55                      cmdflag[i]=1;
56              }
57              else
58                      *p++=c;
59        }
```

```
60        *p='\0';
61        return(++i);
62      }
63
64      docommand(int i)
65      {
66        int j, stat, pid;
67        char *argl[MAXARG], args[LINSIZ];
68        char c, *argsp, **arglp, *p;
69
70        for(j=0;j<i;j++)
71        {
72            arglp=argl;
73            argsp=args;
74            p=cmdbuf[j];
75            while((c=*p++)!='\0')
76            {
77                while(c==' '|| c=='\t')
78                    c=*p++;
79                if(c=='\0')
80                {
81                    *argsp++='\0';
82                        break;
83                }
84                *arglp++=argsp;
85                while(c!=' '&&c!='\t'&&c!='\0')
86                {
87                    *argsp++=c;
88                    c=*p;
89                    if(c)    p++;
90                }
91                *argsp++='\0';
92            }
93            *arglp=(char *)0;
94            if(strcmp(argl[0],quit)==0)
95            {
96                 printf("Bye Bye!\n");
97                exit(0);
98            }
99            if((pid=fork())==0)
100           {
101               if(cmdflag[j]) setpgrp();
102               execve(argl[0],argl,environ);
```

```
103                             printf("Returned from execve: %s\n",cmdbuf[i]);
104                             exit(10);
105                     }
106                     else
107                     {
108                             if(! cmdflag[j])
109                                     while(wait(&stat)!=pid);
110
111                     }
112             }
113     }
```

程序代码说明：

- 数据结构说明：该 mini_SH 定义了每条命令所能使用的最大参数个数为 MAXARG，定义为 10(第 12 行)，除第一个为命令本身，最后一个为空外，用户提交命令一次所带的参数不能大于 8。每条命令的字符缓存数组由 LINSIZ 决定，最大为 80(第 13 行)，一次提交的命令个数由 CMDSIZ 决定，最多 8 个(第 14 行)。字符指针 quit 存放退出命令字符串，它已赋值为 quit.quit(第 17 行)，二维数组 cmdbuf 存放从标准输入读到的命令串(第 18 行)，而数组 cmdflag 决定该命令以何种方式执行(第 19 行)，0 为前台方式，1 为后台方式。
- main 函数：按主流程的设计思想实现，在无限循环 for 中(第 24 行)，首先打印命令接收提示符 mini_SH-->(第 26 行)，将用户键入的命令字符串，通过函数 readcmd 得到并译码到已定义的外部变量 cmdbuf 二维数组中(第 32 行)，readcmd 返回一次提交的用分号隔开的命令个数。而函数 docommand 执行存放在 cmdbuf 中的命令(第 33 行)。
- readcmd 函数：将用户从标准输入提交的一行命令，按分号为界，分别存入命令缓冲区 cmdbuf 中(第 44~59 行)，如果命令字符中有‘&’符，将命令标志数组 cmdflag 的相应值置为 l(第 53~56 行)，每次命令提交时，该数组字段被清为零(第 60 行)。该函数返回该次用户提交的命令个数(第 61 行)。
- docommand 函数：在 for 循环中(第 70 行)，每次执行一条命令。用户提交的命令按顺序存放在数组 cmdbuf 中，while 循环将命令执行的参数以空格或制表符为分界线，将字符型的数组指针 argl 分别指向相应的字符串，argl[0]指向该命令字符串，argl[1]是该命令的第一个参数，依此类推，最后一个参数为一空指针。对于每个参数增加一个空结束符‘\0’(第 75~92 行)。如果命令字符串为我们定义的退出该命令的字符串 quit.quit，则调用 exit 系统函数退出执行(第 94~98 行)，否则调用 fork 生成子进程(第 99 行)。设置后台标志位的话，重新设置进程组号(第 101 行)，使用带环境变量的系统调用 execve 执行用户提交的命令(第 102 行)，如该命令以后台方式执行，则父进程不等待该命令执行完就可执行新的命令(第 103~104 行)，否则父

进程使用 wait 函数等待子进程执行暂停或终止(第 109 行)。

该程序是一个完整的程序，编译后生成 mini_sh 命令，以下是在 mini_sh 命令控制下，用户提交命令的执行情况：

```
$ ./mini_sh
mini_SH-->/bin/date                          /* 输入命令 */
2008 年 03 月 25 日 星期二 21:14:52 CST
mini_SH-->/bin/who                           /* 输入命令 */
Returned from execve:
mini_SH-->/bin/pwd;/bin/date                      /*同时输入 2 个命令 */
/home/lxy/test1/chapter6
2008 年 03 月 25 日 星期二 21:15:17 CST
mini_SH-->quit.quit                          /* 退出 */
Bye Bye!
```

在 mini_SH 控制之下的命令是可以带 8 个参数执行的，也可以后台形式执行，但由于我们使用的是 execve 系统函数，因此对命令而言，除当前目录下可用路径名分量外，其他都必须为全称路径名。但通过使用 execvp 函数可以修改这一点，感兴趣的读者可通过修改该实例，用 execvp 系统函数代替 execve 来验证这一点。

例 5-15 该程序的功能为，执行从命令行中一次提交的多条不带参数的命令，并且分析进程终止的原因。程序代码如下：

```
1        /* ex15.c*/
2        #include <sys/types.h>
3        #include <unistd.h>
4        #include <stdio.h>
5
6        int main(int argc, char *argv[])
7        {
8          int i, pid, stat;
9          char *p;
10         if(argc<=1)
11         {
12                printf("Usage %s cmd1 cmd2 ... \n",argv[0]);
13                exit(1);
14         }
15         setbuf(stdout,NULL);
16         for(i=1,p=argv[1];argc>1;p=argv[1])
17         {
18                printf("===[%d]:command '%s' begin ===\n",i,p);
```

```
19                  if((pid=fork())==0)
20                  {
21                          printf("Child pid =%d\n",getpid());
22                          execl(argv[1],argv[1],0);
23                          exit(argc);
24                  }
25                  pid=wait(&stat);
26                  statport(pid,stat);
27                  argc--;argv++;
28                  printf("===[%d]:command '%s' end === \n\n",i++,p);
29          }
30      }
31
32      struct sigway
33      {
34        char *sigstr;
35        int value;
36      };
37
38      struct sigway sigways[]=
39      {"hup",1, "int",2, "quit",3, "ill",4, "trap",5, "iot",6, "abrt",7, "emt",8,    "fpe",9, "kill",10,
        "bus",11, "segv",12, "sys",13, "pipe",14, "alrm",15, "gterm",16, "user1",17, "user2",18,
        "cld",19, "pwr",20, "poll",21,  (char *)0,0 };
40
41      statport(int pid, int stat)
42      {
43        int i;
44        if(pid==-1)
45        {
46              printf("bad wait\n");
47        }
48        else if((stat&0177)==0177)
49        {
50              i=stat>>8;
51              printf("child process: %d stop by signal\n",pid);
52              printf("Signal name: %s\n",sigways[i].sigstr);
53              printf("Signal value: %d\n",sigways[i].value);
```

```
54          }
55          else if((stat&0xff)==0)
56          {
57                  printf("Child process: %d exit by 'exit' system call.\n",pid);
58                  printf("exit code: %d\n",stat>>8);
59          }
60          else if((stat>>8)==0)
61          {
62                  if((stat&0200)==0200)
63                          printf("child process: %d '-core dumped' by signal \n",pid);
64          }
65      }
```

该程序首先检查命令行中输入命令个数是否满足要求(第 10~14 行)，根据命令行提交的命令个数执行循环体中的程序段，for 循环中的程序先调用 fork 创建子进程(第 19 行)。子进程打印出自己的进程号，然后用 exec 系统函数执行用户在命令行中提交的命令，如 execl 执行失败，则调用 exit 向父进程传递参数，argc 后终止(第 21~23 行)。父进程在 fork 后调用 wait 等待子进程暂停或终止。父进程被唤醒后，再调用函数 statport 来分析并输出子进程暂停或终止的原因(第 25~26 行)，然后修正命令行参数值，执行下一条命令(第 27 行)。

在函数 statport 中，参数为子进程 ID 号和子进程退出状态代码，程序根据退出状态代码输出相应的暂停或终止原因(第 44~63 行)。

下面是该程序的两种执行情况：

(1) 命令行中的参数为 3 个系统命令。

```
$./ex15 /bin/date ls /usr/bin/who
===[1]:command '/bin/date' begin ===
Child pid =8007
2008 年 03 月 25 日 星期二 22:16:03 CST
Child process: 8007 exit by 'exit' system call.
exit code: 0
===[1]:command '/bin/date' end ===

===[2]:command 'ls' begin ===
Child pid =8008
Child process: 8008 exit by 'exit' system call.
exit code: 3
===[2]:command 'ls' end ===

===[3]:command '/usr/bin/who' begin ===
```

```
Child pid =8009
lxy          tty7             2008-03-25 08:39 (:0)
lxy          pts/0            2008-03-25 08:47 (:0.0)
Child process: 8009 exit by 'exit' system call.
exit code: 0
===[3]:command '/usr/bin/who' end ===
```

从结果可看到第一条命令和第三条命令是使用隐含的 exit(0)系统函数退出的，它们传给 wait 系统函数的值为 0，因为调用 execl 执行这两条命令是成功的。而调用 execl 执行第二命令时失败了(原因是没有使用绝对路径/bin/ls)，传递给父进程的状态值为命令行参数个数 argc。

(2) 命令行中的参数包含用户编写的程序。

```
./ex15 ./test1 /bin/date
===[1]:command './test1' begin ===
Child pid =8061
child process: 8061 '-core dumped' by signal
===[1]:command './test1' end ===

===[2]:command '/bin/date' begin ===
Child pid =8064
2008 年 03 月 25 日 星期二 22:20:34 CST
Child process: 8064 exit by 'exit' system call.
exit code: 0
===[2]:command '/bin/date' end ===
```

其中，命令行中的参数 test1 是用户自编程序，它的源程序为：

```
/*test1.c*/
int main()
{
 char *p;
 *p='c';
}
```

execl 系统函数更换进程映象执行 test1 后，因对空指针赋值接收到一个信号而自行终止，并产生一个 core 文件。有关信号的讨论参见下一章。

5.4 小 结

本章介绍了操作系统中最重要的一个概念：进程。介绍了 Linux 系统中与进程有关

的一些系统函数，包括进程的创建、等待、结束等操作，进程的用户标识号管理和进程标识号管理等。完整的理解 Linux 进程操作是非常重要的，读者需要对以上系统函数熟练掌握。

习　　题

一、填空题

1. 进程在其生存期内可能处于三种基本状态：______、______、______。

2. 为了让 Linux 来管理系统中的进程，每个进程用一个______数据结构来表示。

3. 在 Linux 系统中，进程有两种运行模式：______和______。

4. 创建一个新进程的唯一方法是由某个已存在的进程调用______或______函数，被创建的新进程称为______，已存在的进程称为______。

5. 系统中的每个进程都有唯一的非负整数作为其标识，它被称为______。

二、选择题

1. 可运行进程是一个只等待______资源的进程。进程可以忽略大部分信号，但下列信号中______是不能忽略的。

(A) 内存 (B) CPU (C) 键盘 (D) 终端

2. fork 函数在父进程中的返回值是______。

(A) 创建的子进程的进程标识号 (B) 0 (C) -1 (D) 1

3. 在 Linux 中，下列不属于正常结束进程的方法是______。

(A) 在 main 函数中调用 return (B) 调用 exit 函数

(C) 调用_exit 函数 (D) 调用 abort 函数

4. 一个进程调用 wait 或 waitpid 函数，可能产生 3 种情况，下列不属于这 3 种情况的是______。

(A) 如果所有子进程都还在运行，进程挂起

(B) 如果恰有子进程结束，它的终止状态字正等待父进程提取，立即得到该终止状态字并返回，其返回值为该子进程的进程号

(C) 如果该进程没有子进程，立即返回，返回值为-1

(D) 如果该进程没有子进程，立即返回，返回值为 0

5. 返回调用进程的进程标识号的系统函数是______。

(A) getpid (B)getpgrp (C) getppid (D) setpid

三、上机题

1. 编写一个程序，在主进程中创建一个子进程，子进程输出“Hello World！”字符串后退出，然后主进程退出。

2. 编写一个程序，在主进程中创建一个子进程，子进程进行空循环，不停地输出“Hello World！”字符串，主进程休眠一段时间后，在主进程中结束子进程，随后主进程也退出。用命名管道实现客户到服务器之间传递数据的操作。

3. 编写一个程序，在程序中使用命令行形式显示程序所在当前文件夹下的内容。

4. 编写一个程序，得到当前进程的标识号，并将它打印输出，随后写入一个文件中。

CHAPTER 6

进程间通信(IPC)

复杂的编程环境通常使用多个相关的进程来执行有关操作。进程之间必须进行通信来共享资源和信息。因此要求内核提供必要的机制，这些机制通常称为进程间通信，或IPC(InterProcess Communication)。

进程间通信有如下一些目的：

(1) 数据传输。进程可能要发送数据到另一个进程。发送的数据量可以在一个字节到几兆字节之间。

(2) 共享数据。多个进程想要操作共享的数据。一个进程修改了数据，其他共享该数据的进程应该立即看见这个变化。

(3) 通知事件。当一些事件发生时，进程也许会向另一个进程或一组进程发消息通知事件的发生。比如，进程终止时，它要通知它的父进程。接收者可能是被异步通知的，这时候它的正常处理被中断。由此，接收者可以选择等待通知。

(4) 资源共享。一些要求相互操作的进程需要自行定义一些协议，这些协议针对它们要访问的特定的资源。这些协议是通过使用锁和同步机制来实现的，而锁和同步机制是建立在内核提供的基本功能之上的。

(5) 进程控制。有些进程，比如 debugger 希望完全控制另一个进程(目标进程)的执行。控制进程希望能够拦截为目标进程设计的所有的陷入和异常，并且能够及时知道目标进程的状态改变。

6.1 进程间通信机制概述

进程间的通信机制(IPC)，就是多进程间相互通信、交换信息的方法。Linux 支持多种

IPC 机制。信号与管道是其中的两个，Linux 还支持传统的 Unix System V 的 IPC 机制。

6.1.1 信号

信号主要用来通知进程异步事件的发生。最初信号设计的目的是为了处理错误，它们也用来作为最基本的 IPC 机制。在 Linux 中可以识别 64 种不同的信号。这些信号中的大部分都有了预先定义好的意义，但是至少有两个，SIGUSR1 和 SIGUSR2 可以由应用程序来定义。进程可以显式地用 kill 或是 killpg 系统函数来向另一个进程或进程组发信号。此外，内核可以响应不同的事件而产生内部信号。例如，当在按下终端键“Ctrl+C”时，内核便发送一个 SIGINT 信号到前台的进程。

每一个信号都有一个默认的动作，典型的默认动作是终止进程。进程可以通过提供信号处理函数来取代对于任意信号的默认反应。信号发生时，内核中断当前的进程，进程执行处理函数来响应信号，信号处理完后，进程恢复正常的处理。这就是事件通知进程和进程响应异步事件的方式。信号也可以用来同步，进程可以调用 sigpause 以等待信号的到来。

信号最初设计目的主要是来处理错误。例如，内核把一些硬件异常错误，例如被零除或其他无效指令等转换成信号，如果进程没有对这些异常的处理程序，则内核要处理这些错误，通常的做法就是终止进程。

作为一种 IPC 机制，信号有一些局限性——信号的系统开销太大。发送信号的进程要进行系统调用；内核要中断接收信号的进程，而且要管理它的堆栈，同时还要调用处理程序，之后还要恢复执行被中断的进程。更重要的是，信号的数量非常有限，因为只存在有限的不同的信号，而且信号能传送的信息量十分有限，用户产生的信号不可能发送附加信息及各种参数。信号对于事件的通知很有用，但是对于复杂的交互操作，信号是不能胜任的。

系统中预定义了一些信号，这些信号可以由内核产生，也可以由一些有特殊权限的进程产生。可以使用 kill -l 命令显示系统中的信号，以下是在笔者的计算机上运行上述命令后显示的信号：

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
```

```
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6
59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

进程可以忽略大部分信号，但是有两个是不能忽略的。

(1) SIGSTOP：这个信号将中断进程的执行。

(2) SIGKILL：这个信号将强制进程退出。

除了这两个信号外，对于其他的信号，进程可以选择处理某些信号。进程可以阻塞(block)某些信号，所谓阻塞某些信号就是信号的发生对这个进程不会产生任何影响，进程也不会做某些动作响应这个信号。如果进程不阻塞信号，那么可以选择由进程自己来处理还是由内核来处理这些信号。如果由内核处理这些信号，那么就会调用标准信号处理程序。例如，当一个进程收到一个 SIGFPE(浮点溢出错误)时，系统默认的处理是 core dump(内核转储)，然后退出。信号间并没有优先级的差别，例如，同时发送给一个进程的两个信号可能以任何顺序由进程来处理。另外，系统将多个同一类型的信号当作一个来处理，例如，一个进程处理一个 SIGCONT 和 10 个 SIGCONT 的方法是一样的。

除了 SIGSTOP 和 SIGKILL，其他的信号都可以被阻塞。如果系统产生了一个阻塞的信号，那么这个信号将一直处于待处理状态直到非阻塞(unblocked)状态为止。

为了让进程处理信号，Linux 必须维护进程如何处理信号的信息，这些信息保存在一个 sigaction 数据结构中。在 sigaction 结构中，保存了处理信号的例程的地址，或者保存了一个标志位，通过这个标志位，Linux 内核可以知道是忽略这个信号还是由内核来处理这个信号。通过系统调用，进程可以改变信号的默认处理方式，而这种改变就是通过改变 sigaction 结构来实现的。

并不是系统中的所有进程都可以发送信号到其他进程，实际上，只有内核和超级用户有上述权限。普通进程只能向那些有相同 UID 和 GID 的进程发送信号，或者向那些在一个进程组(process group)中的进程发送信号(父进程和其所有的子进程的集合叫做一个进程组，在 Linux 中，默认的进程组是属于同一个登录 shell 的所有进程。每一个进程组有一个唯一的 ID 号，叫做 GID，一个进程组的 GID 是由创建进程组的进程的 PID 所决定的，通常创建进程组的进程就是登录的 shell 进程)。通过设置进程 task_struct 结构 signal 成员的某些位可以产生各种信号。如果进程没有阻塞一个信号，而且进程处于等待状态，那么进程就可以被信号唤醒并处于运行态，进程调度会把这个进程放在可执行队列中。如果一个信号需要默认的处理程序，Linux 的信号处理程序会使用经过优化的方法，通常这种优化的方法是处理信号的最佳方法，此时进程不需要做任何事情。

信号在产生后并不是立即发送给进程的，它必须等到进程再次运行时才能发送。每当进程从系统调用中返回时，系统就检查进程的 task_struct 结构，如果有任何非阻塞的信号，那么就发送这个信号。进程也可以等待特定的信号，在信号出现之前，进程处于挂起状态。

如果一个信号处理程序被设置为默认的动作，那么内核会处理这个信号。例如，

SIGSTOP 信号的默认处理程序将改变当前进程的状态为停止状态，然后进程调度会选择一个新进程来运行。SIGFPE 信号的默认动作是内核转储，然后强迫进程终止。

除了默认的信号处理以外，进程也可以指定一个自己的信号处理程序，这才是编程需要处理的问题。当信号产生时会调用相应的信号处理程序。在 sigaction 结构里保存了信号处理程序的地址。

Linux 使用了堆栈来管理要执行的信号处理程序，这样当一个信号处理程序完成操作时，下一个将被调用，依次下去。

6.1.2　管道

在传统的实现中，管道是单向的、先入先出的、无结构的、固定大小的数据流。写进程是在管道的尾端写入数据，读进程是从管道的首端读出数据。数据读出后，将从管道移走，其他读进程都不能再读到这些数据。管道提供了简单的流控制机制。进程试图读空管道时，在有数据写入管道前，进程一直阻塞。同样，管道已经满时，进程想写入数据，在其他进程从管道中读走数据(也就是移去数据)之前，这个写进程将发生阻塞。

系统调用 pipe 生成一个管道并返回两个描述符，一个用来读管道，一个用来写管道。这些描述符为子进程所继承，因此它们可以共享访问文件。通过这种方式，每个管道可以有许多读进程和写进程。给定的进程可以是读进程也可以是写进程，或者两者都是。

然而，通常管道被两个进程共享，每个进程拥有管道的一端。对管道的 I/O 操作和对文件的 I/O 操作非常相似，通过对管道的描述符调用 read 和 write 系统调用来操作。进程通常不知道它正在读或写的实际上是一个管道。

典型的应用程序，如 shell 控制管道描述符，从而管道仅仅有一个读进程和一个写进程，因此使用管道作为单方向的数据流。最普通的管道应用就是让一个程序的输出变为另一个程序的输入，用户通常使用 shell 的管道操作符“|”来连接两个程序。

从 IPC 角度看，管道提供了从一个进程向另一个进程传输数据的有效方法。但是，管道有一些固有的局限性：

(1) 因为读数据的同时也将数据从管道移去，因此管道不能用来对多个接受者广播数据。

(2) 管道中的数据被当做是字节流，因此无法识别信息的边界。如果写进程发送不同长度的数据对象通过管道，那么读进程不能确定接收了多少个对象，或是它不能确定对象的边界。

(3) 如果一个管道有多个读进程，那么写进程不能发送数据到指定的读进程。同样，如果有多个写进程，那么没有方法来判别是它们中的哪一个发送了数据。

我们知道，Linux 的 shell 可以使用重定向，例如下面的例子：

```
$ ls | cat
```

上面命令的执行过程是这样的：我们知道 cat 是用来显示文件内容的，ls 是列出文件目录内容。上面的 2 个命令通过管道连接起来，管道前面的命令的输出是管道后面命令的输入。ls 列出目录内容，并将目录内容送到 cat，cat 将目录内容进行显示。上面所说的重定向，是指通过管道将命令的输入和输出重新定向到其他地方，而不是标准输入和标准输出。经过重定向的命令并不知道已经被重定向了，所以它同处理标准输入输出一样完成任务。使用管道和重定向技术可以实现许多功能的扩展，在后面我们会通过例子来说明如何通过管道来重定向一个进程的标准输入和输出。

在 Linux 中，管道是通过两个 file 数据结构来实现的，这两个结构指向同一个临时 VFS 的 i 节点(这里指的是创建非命名管道的情形，关于命名管道和非命名管道我们会在后面详细讨论)，这个 i 节点指向内存中的一个物理页面。这两个 file 结构分别属于写入和读出进程，它们包含了指向不同的文件操纵例程的指针向量，一个是用来向管道中做写操作，另一个用来从管道中读，通过这种方法可以隐藏用于操纵文件的底层细节，因为我们可以像操纵文件一样来处理管道。当写进程向管道中写数据时，这些数据会被复制到共享数据页面中，当读进程从管道中读数据时，数据将从共享数据页面中复制出来。另外，Linux 必须同步对管道的访问，它必须保证对管道的读写是按次序完成的，即首先写进程向管道中写数据，然后读进程从管道中读数据。

当写进程想要往管道里写数据时，它使用标准的 write 库函数。写操作将文件描述符传递给进程的 file 结构，每一个 file 结构相当于一个打开的文件(或者说一个打开的管道)。Linux 的系统调用使用描述管道的 file 结构指向的写例程来完成向管道中写数据的操作，这样只要使用标准的处理文件的系统调用就可以操纵管道了。在代表这个管道的 VFS 的 i 节点中保存有写例程用来管理写操作时所需的信息。如果共享页面中有足够的空间来写入数据，而且管道没有被其他的读进程锁定，那么 Linux 首先将这个管道锁定，然后从进程的地址空间中将数据复制到共享数据页面中。如果管道被某一个读进程锁定，或者在共享页面中没有足够的空间，那么当前进程将被放到管道的 i 节点等待队列中。因为管道是可中断的，所以它可以接收信号。当读进程完成了读的工作后，管道将被解锁，或者当共享页面中有足够的空间时，管道可以被读进程唤醒。当写操作完成后，管道的 VFS 的 i 节点将解锁，在 i 节点的等待队列中的处于睡眠状态的读进程将被唤醒。

从管道中读取数据的过程与写数据的过程基本一样。读进程从管道中读数据，如果管道中没有数据，或者管道锁定了，那么就返回一个错误。另外读进程也可以在管道的 i 节点等待队列中排队，直到写进程完成了操作时才被唤醒。当两个进程都完成了对管道的操作后，管道的 i 节点连同共享数据页面将被释放。上面的整个过程是典型的非命名管道的读/写流程，非命名管道的生命周期就是读/写进程进行管道操作的时间，它是一个临时的对象。

Linux 还支持一种称为命名管道(named pipe)的机制，命名管道也叫 FIFOs，因为管道的操作原则是 First In，First Out。第一个向管道中写入的数据同样将是第一个从管道中读取的数据。但是，与非命名管道不同的是，FIFOs 并不是一个临时的对象，它们是文件系

统中的一个实体，可以通过 mkfifo 命令来创建。如果进程有适当的权限访问 FIFO，那么在使用 FIFO 时相当自由。FIFOs 打开的方式与非命名管道有一些差别，因为非命名管道是临时的对象，在使用之前必须创建它，而 FIFO 是一个已经存在的对象，所以只要打开并使用它就可以了。在使用命名管道时，Linux 必须首先处理读进程的打开操作，然后再处理写进程的打开操作。同样，Linux 首先处理读进程的读操作，然后处理写进程的写操作。除了上面的几点不同之外，命名管道与非命名管道的处理方法都是一样的，而且它们使用相同的数据结构和操作方法。

6.1.3 System V IPC 机制简介

Linux 支持 Unix System v 中的三种进程间通信机制，它们是：消息队列、信号量(或者称信号灯)和共享内存。这几个 System V IPC 机制有一个共同的特点，就是它们使用相同的认证方法，一个进程只有通过系统调用向内核传递一个唯一的引用标识符才能访问这些资源。访问这些 System V IPC 对象首先要经过权限检查，就如同访问文件要经过权限检查一样。这几个 System V IPC 对象的访问权限是对象创建者通过系统调用设定的。对象的引用 ID 是资源表中的一个索引，这个索引并不是直接的索引，而是要经过一些操作而产生的索引。

在 Linux 中，描述这几种 System V IPC 对象的数据结构中都包含了一个 ipc_perm 结构，这个结构中包含了对象的所有者、创建者和进程的用户 ID 及组 ID，还包括对象的访问权限(分别对所有者、同组人和其他用户)和 IPC 对象关键字(或者称键值)。关键字用来确定 System V IPC 对象的引用 ID 的位置。Linux 中支持两类关键字：public 和 private。如果关键字是 public，那么系统中的所有进程都可以找到 System V IPC 对象的引用 ID。如果关键字是 private，那么只有对象的创建者和同组人有权查看 System V IPC 对象的引用 ID。另外，请注意 System V IPC 对象只能通过它们的引用 ID 来引用，不能通过关键字来引用。

System v IPC 资源有一些统一的属性，总结如下：

(1) 键(key)：一个由用户提供的整数，用来标志这个资源的实例。

(2) 创建者(creator)：创建这个资源的进程的用户 ID(UID)和组 ID(GID)。

(3) 所有者(owner)：资源的所有者的 UID 和 GID。资源创建的时候，资源的创建者就是资源的所有者。拥有能改变所有者权力的进程可以给资源指定一个新的所有者。资源的创建者进程、当前的所有者进程和超级用户具有这个权力。

(4) 权限(permissions)：文件系统类型的权限。指资源的所有者进程，同组中的进程和其他用户对资源的读/写/执行的权限。

6.1.3.1 消息队列

消息队列就是消息链表的头部指针。消息队列允许一个或多个进程写消息，同样这个消息可以被一个或多个进程读取。消息队列中的消息也是一个结构，称为 msg，它通常包括一个 32 位的类型值，其余的是数据区域。Linux 维护了一个消息队列的链表，叫做 msgque，

链表中的每一个元素指向一个 msqid_ds 数据结构，msqid_ds 结构用来描述消息。当消息队列创建时，Linux 在系统内存中分配一个叫 msqid_ds 数据结构的空间，并将它加到消息链表中。

每一个 msqid_ds 结构包含一个 ipc_perm 结构和一个指向消息的指针。另外，Linux 还要记录队列的修改时间等信息。msqid_ds 还包含两个等待队列，一个是写等待队列，另一个是读等待队列。

当进程试图向队列中写消息时，系统将其有效 UID 和有效 GID 同队列的 ipc_perm 结构中的 mode 成员进行比较，如果进程有写的权限，那么这个消息将从进程的地址空间复制到 msg 结构中，并放在消息队列的末尾。每一个消息都被标记为与应用程序相关的某种类型，这些类型在相互协调的进程间有统一的意义。Linux 对消息的数量和长度都有限制，如果系统中没有足够的空间容纳新消息，那么要发送消息的进程将被加到消息队列的写等待队列中，当系统中有消息被读取时，即消息队列有足够的空间容纳新消息时，这个进程将被唤醒。

读取消息的过程与写消息的过程类似，一个读取进程可能选择队列中的第一个消息，而不管其类型如何，或者读取具有某种类型的第一个消息。如果没有消息符合读取进程的选择标准，那么这个读取进程将被加到消息队列的读等待队列中。当一个新消息写到消息队列中后，这个进程将被唤醒，重复上面的工作。

6.1.3.2　信号量(semaphores)

信号量是为了控制进程对资源的使用而发明的。信号量是具有整数值的对象，它的工作原理如下：它支持两种原子操作 P 和 V，P 操作减少信号量的值，如果某一个信号量的值小于 0，则操作阻塞；V 操作增加信号量的值，如果结果值大于或等于 0，V 操作就要唤醒一个等待进程。这两个操作是原子的，即它们是最小操作，是不可分割的。

信号量可以用来实现一些同步协议。比如，考虑管理一个计数资源，也就是说资源有固定数目的实例。进程想获得资源的一个实例，当它使用完这个资源后释放它，这个资源能用初始化的数值实例的信号量表示。想获得资源时使用 P 操作，每次请求成功，它都要减少信号量的值。信号量的值减至 0 时(无空闲可用资源)，下一个 P 操作将被阻塞。释放资源的时候使用 V 操作，它增加信号量的值，同时唤醒被阻塞的进程。

一个最简单的信号量就是内存中的一个地址，这个地址中的值可以被其他的进程检查或设置。检查/设置操作是不能中断的，换句话说是原子(atomic)操作，一旦启动任何程序都不能终止它。检查/设置操作的返回结果是信号量当前的值与设置值(可正可负)之和。根据检查/设置操作的返回值的不同，一个进程可能要处于睡眠状态直到信号量的值被其他进程改变了。用信号量可以实现临界区的概念，一个临界区是一段代码，在任意时刻只能有一个进程执行它。

假设有许多进程要从一个文件中读取记录并将记录写入到文件中，例如数据库管理程序就要经常做上面的操作，那么这些读/写操作要互相协调。这种情况下可以使用一个初始值为 1 的信号量，在文件操作的代码中，使用两个信号量操作，一个增加信号量的值，

另一个减少信号量的值。访问文件的第一个进程会减少信号量的值并成功返回，此时信号量的值为 0，然后这个进程就可以继续对文件进行其他的操作，如果在操作的过程中，有另外一个进程想访问文件，同样它会减少信号量的值，但是返回失败，因为信号量的值将为-1。这个进程会阻塞并等待第一个进程完成操作。当第一个进程完成操作时，它会将信号量的值加 1，这样信号量值又恢复为原来的 1。现在，那个等待的进程将被唤醒，因为此时信号量的值为 1，所以它的减小信号量值的操作将成功返回。

6.1.3.3 共享内存

共享内存区域是被多个进程共享的一部分物理内存。进程可以把这些区域映射到它们地址空间中的任一合适的虚拟地址范围。这些地址范围对每一个进程来说可以是不同的。映射后，这些区域就可以像其他任何内存位置那样被访问，而不需对它使用读(read)或写(write)调用。因此，共享内存机制提供了进程共享数据的最快方法。进程向共享内存区域写入了数据，那么共享这个区域的所有进程可以立即看见共享区域中新的内容。图 6-1 描述了两个进程共享一块内存区域的情形。

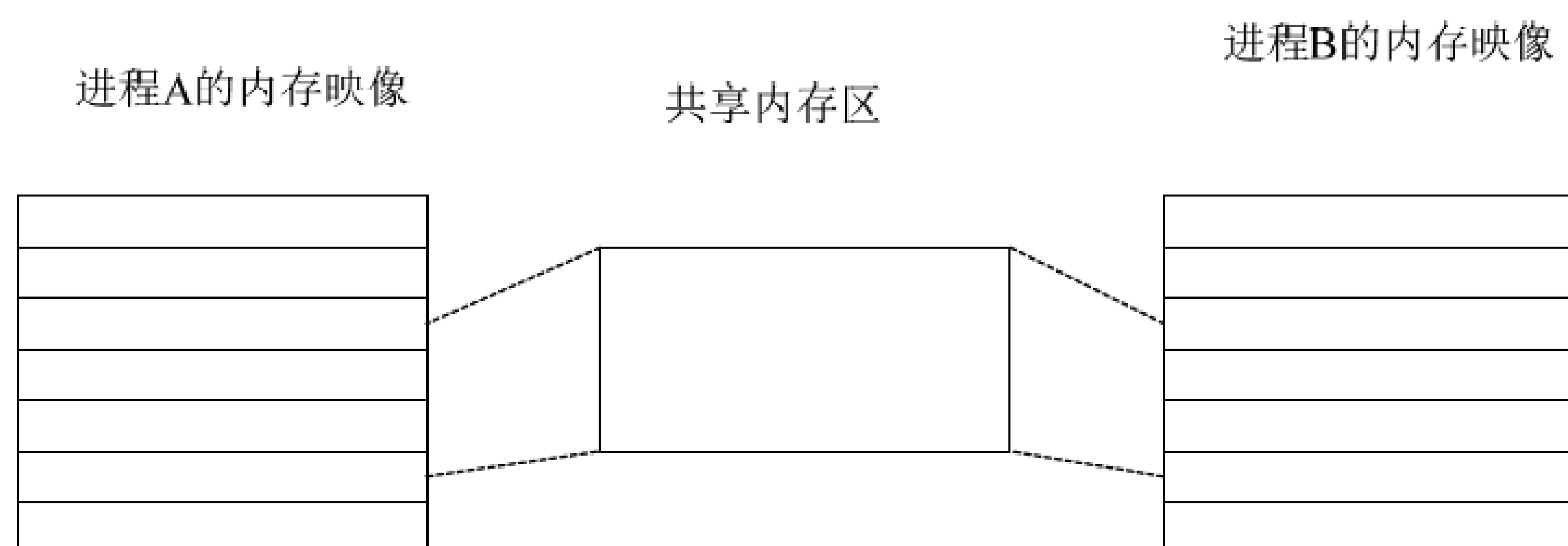


图 6-1 进程间共享内存示意图

共享内存允许多个进程通过一块在这些进程的虚拟地址空间之间共享的内存来通信。虚拟内存的页面是通过每一个共享进程的页表的页表项来引用的。如同访问所有 System V IPC 对象的方法一样，访问共享内存要通过关键字和访问权限来控制。一旦内存共享出来，Linux 就不检查进程是如何使用这个共享内存的了。所以，要安全地使用共享内存，必须依靠其他机制，例如使用信号量等来同步访问共享内存。

共享内存的访问权限和关键字是由共享内存的创建者来设置的。如果创建者有足够的权限，还可以在物理内存中锁定共享内存。

共享内存提供了一种快速灵活的机制，允许不用复制或是使用系统调用的方法就可以共享大量的数据。它的主要局限性就是它不能提供同步。如果两个进程企图修改相同的共享内存区域，内核不能串行化这些操作，因此写入的数据可能任意地互相混合。使用共享内存的进程必须设计它们自己的同步协议。为了实现这些同步操作，进程必须进行额外的操作，这样会对共享内存的性能有所影响。

6.2 信 号 处 理

Linux 是一个多用户、多任务的操作系统，无论是操作系统与一般进程间的通信，还是用户进程间的通信都是必要的。信号是进程间互相通信的方法之一，它用来指出某种事件的发生。

在 Linux 系统中，针对不同的软硬件状况，内核程序会发送出不同的信号来通知进程某个事件的发生。但是如何处理这个信号，就要由进程本身来处理。

信号可以由系统内核程序发出，也能由某些进程发送，但是大部分的时候都是由内核程序发出的。

当一个信号正在被处理时，所有同样的信号都将暂时搁置，直到这个信号处理完成。

进程接收到核心程序所发出的信号后，处置的方式有下面几种：

(1) 忽略这个信号。

(2) 执行一个处理此信号的函数。

(3) 暂停进程的执行。

(4) 重新启动刚才被暂停的那个进程。

(5) 采用系统默认的行动。大部分信号的默认操作都是终止进程的执行。

有些信号除了会终止进程的执行，还会留下一个叫 core 的文件。这个过程叫做内核转储(core dump)，这个文件存有进程当时在内存中的内容，通常用于事后查错。

6.2.1 信号类型

前面介绍了可以用 kill-l 查看系统中预定义的信号。下面详细介绍一些信号的意义。

Linux 支持 POSIX.1 中的信号，表 6-1 列出了这些 POSIX.1 的信号及其默认动作和解释。注意有些信号是与 CPU 类型和系统体系结构有关的。

表 6-1 POSIX.1 支持的信号

信 号	值	动 作	意 义
SIGHUP	1	A	挂断控制终端，当一个终端被切断时，核心程序就将此信号传给该终端所控制的一切进程
SIGINT	2	A	控制终端的中断键被按下
SIGQUIT	3	A	从键盘中断中退出
SIGILL	4	C	不正确的硬件指令，应用程序通常会捕获此信号以响应程序执行时的错误
SIGABRT	6	C	调用 abort 系统函数放弃信号

(续表)

信　　号	值	动　　作	意　　义
SIGFPE	8	C	浮点溢出错误
SIGKILL	9	AEF	删除一个或一组进程，本信号不能忽略
SIGSEGV	11	C	不合法的内存引用
SIGPIPE	13	A	断开的管道：一个进程不停地将数据写入管道，但是没有进程读数据，即读管道的进程非正常终止了
SIGALRM	14	A	时钟，用以测量进程的真实时间(不是 CPU 时间)。alarm 系统调用就是用来设定此信号
SIGTERM	15	A	终止进程。kill 系统调用就发送这个信号
SIGUSR1	30,10,16	A	用户自定义信号
SIGUSR2	31,12,17	A	用户自定义信号
SIGCHLD	20,17,18	B	子进程暂停或终止
SIGCONT	19,18,25		如果进程暂停，那么继续执行
SIGSTOP	17,19,23	DEF	暂停进程
SIGTSTP	18,20,24	D	把停止信号送给联机会话进程，通常由 Ctrl+Z 来产生此信号
SIGTTIN	21,21,26	D	后台执行中的进程要从控制终端读取数据
SIGTTOU	22,22,27	D	后台执行中的进程企图对控制终端写入数据

表 6-2 是 Linux 支持的一些其他信号。

表 6-2　Linux 支持的其他信号

信　　号	值	动　　作	意　　义
SIGTRAP	5	CG	程序跟踪中断点。这是一种给调试程序(例如 gdb)专用的信号
SIGIOT	6	CG	输入输出中断点，通常是由于硬件故障
SIGEMT	7,-,7	G	硬件仿真程序捕俘
SIGBUS	10,7,10	AG	总线错误
SIGSYS	12,-,12	G	系统调用参数错误(SVID)
SIGSTKFLT	-,16,-	AG	协处理器堆栈错误
SIGURG	16,23,21	BG	这个信号通知系统有要求立即处理的 SOCKET
SIGIO	23,29,22	AG	I/O 操作可以执行，例如可以对某个文件描述字进行操作
SIGPOLL		AG	与 SIGIO 同义
SIGCLD	-,-,18	G	与 SIGCHLD 同义

(续表)

信　　号	值	动　　作	意　　义
SIGXCPU	24,24,30	AG	进程超出了所设定给它的最大 CPU 使用时限
SIGXFSZ	25,25,31	AG	进程超出了所设定给它的最大文件极限
SIGVTALRM	26,26,28	AG	用以测量进程的虚拟时间(实际被执行进程的时间)
SIGPROF	27,27,29	AG	用以测量进程的概括时间(指虚拟时间加核心程序执行进程实际时间)
SIGPWR	29,30,19	AG	电源故障
SIGINFO	29,-,-	G	与 SIGPWR 同义
SIGLOST	-,-,-	AG	文件锁丢失
SIGWINCH	28,28,20	BG	X Window 窗口改变大小
SIGUNUSED	-,31,-	AG	未使用的信号

说明：

(1) 在表 6-2 中信号值为“-”表示没有此信号。

(2) 在表 6-2 中，每一个信号值分为三列，前面介绍过，信号是与 CPU 相关的，第一个是 alpha 和 sparc 上的信号值，第二个是 i386 和 PowerPC 上的信号值，第三个是 mips 上的信号值。

(3) 信号值 29 在 alpha 上为 SIGINFO/SIGPWR，在 sparc 上为 SIGLOST。

(4) 动作一栏中的字母意义如下。

A：默认的动作是终止进程。

B：默认的动作是忽略信号。

C：默认的动作是内核转储(core dump)。

D：默认的动作是暂停进程。

E：信号不能俘获。

F：信号不能被忽略。

G：不是 POSIX.1 兼容信号。

(5) SIGIO 和 SIGLOST 有相同的信号值，SIGLOST 是在内核中定义的，但是有些应用程序仍旧把信号值 29 当作 SIGLOST。

6.2.2　处理信号的系统函数

当进程收到信号后，怎样处理这个信号多半是由收到信号的那个进程自行决定，除非收到的信号是 SIGKILL 之类只能采取默认行动的信号。Linux 处理信号主要有下面 4 种方式：

(1) 采用系统默认的处置方式。一般而言，进程对信号的默认处置方式都是终止运行。

(2) 忽略该信号。

(3) 暂时搁置该信号。

(4) 由程序设计人员利用系统调用 signal 设定处理信号的函数。

6.2.2.1 注册信号调用函数

要对一个信号进行处理，就需要给出此信号发生时系统所调用的处理函数。可以为一个特定的信号(除去无法捕捉的信号 SIGKIL 和 SIGSTOP 注册相应的处理函数。如果正在运行的程序的原代码里注册了针对某一特定信号的处理程序，不论当时程序执行到何处，一旦进程接收到该信号，相应的调用都会发生。注册的信号处理函数所采用的系统调用为：

```
#include <signal.h>
void (*signal(int signumber, void((*func)(int)))(int);
```

参数 signumber 表示所注册函数针对的信号，其取值为上一节中提到的信号名。参数 func 通常是指向调用函数的函数指针，它指定收到信号后进程所应采取的行动，这便是所谓的信号处理函数。此函数有一个整数参数且返回值为 void。这个信号处理函数可能是用户自定义的一个函数，或是下面的 2 个值：SIG_IGN 和 SIG_DFL。SIG_IGN 表示忽略 signumber 所指出信号。SIG_DFL 表示调用系统定义的缺省处理。信号处理函数的参数是要处理的信号的信号值。另外，不能为 SIGKILL 和 SIGSTOP 设置信号处理函数。signal 函数的返回值类型同参数 func，是一个指向某个返回值为空的带有一个整数参数的函数指针。其正确返回值应为上次信号的处理函数，错误返回-1。

当程序执行 signal 后，表示自此参数 1 的信号将受到参数 2 的函数的管制。注意并非是程序执行到 signal 这一行就立即会对该信号做什么操作。这个道理十分简单，因为信号的产生是无法预期的，程序设计人员根本没法预知该在哪一行捕捉突如其来的信号。用 signal 设置信号处理函数只是告诉系统对这个信号用什么程序来处理。

下面通过一个简单的例子说明 signal 系统调用的使用。

例 6-1 signal 系统调用的使用。

```
1   /*ex1.c*/
2   #include <stdio.h>
3   #include <signal.h>
4
5   void sigcatcher(int signum);
6
7   int main()
8   {
9     char buffer1[100], buffer2[100];
10    int i;
11    if(signal(SIGTERM, &sigcatcher)==-1)
12    {
13            printf("Couldn't register signal hanlder!\n");
14            exit(1);
```

```
15   }
16   printf("Pid of This Process : %d \n",getpid());
17   printf("Please input:\n");
18   for(;;)
19   {
20       fgets(buffer1, sizeof(buffer1),stdin);
21       for(i=0;i<100;i++)
22       {
23           if(buffer1[i]>=97&&buffer1[i]<=122)
24               buffer2[i]=buffer1[i]-32;
25           else
26               buffer2[i]=buffer1[i];
27       }
28       printf("Your input is: %s \n",buffer2);
29   }
30   exit(0);
31 }
32
33 void sigcatcher(int signum)
34 {
35   printf("catch signal SIGTERM.\n");
36   exit(0);
37 }
```

这是一个简单的程序，作用是读入终端输入的字符，并将其中的小写字母转换成大写字母后输出(第 18~29 行)。在这个程序中，注册了针对信号 SIGTERM 的处理函数(第 11~15 行)。因此，当运行这一程序时，程序可以将终端输入的字符串重新输入，直到在另一个窗口中执行 kill pid(pid 是进程 ID，已在程序运行开始时给出)命令向其发送一个 SIGTERM 信号，则转向信号处理函数的调用，输入一条信息后结束进程(第 33~37 行)。运行结果如下：

```
$ ./ex1
Pid of This Process : 7908
Please input:
hello                              /*  输入 hello */
Your input is: HELLO

How are you?                       /*  输入 How are you?   */
Your input is: HOW ARE YOU?

catch signal SIGTERM.              /*在终端的另一窗口输入 kill 7908 */
```

通常情况下，在一个用户进程中需要处理多个信号。可以在一段程序代码中定义对多个信号的处理函数。可以是一个信号对应一个特定处理函数，也可以是多个信号对应同一

个处理函数。下面看一个例子。

例 6-2 可处理多个信号的程序。

```
1   /* ex2.c */
2   #include <stdio.h>
3   #include <signal.h>
4
5   void intfunc(int signum);
6   void exitfunc(int signum);
7
8   int main()
9   {
10    char buffer1[100],buffer2[100];
11    int i;
12    if(signal(SIGINT, &intfunc)==-1)
13    {
14            printf("Couldn't register signal hanlder for SIGINT!\n");
15            exit(1);
16    }
17    if(signal(SIGTSTP, &intfunc)==-1)
18    {
19            printf("Couldn't register signal hanlder for SIGTSTP!\n");
20            exit(1);
21    }
22    if(signal(SIGTERM, &exitfunc)==-1)
23    {
24            printf("Couldn't register signal hanlder for SIGTERM!\n");
25            exit(1);
26    }
27    printf("Pid of This Process : %d \n",getpid());
28
29    for(;;)
30    {
31            printf("Please input:\n");
32            fgets(buffer1, sizeof(buffer1),stdin);
33            for(i=0;i<100;i++)
34            {
35                    if(buffer1[i]>=97&&buffer1[i]<=122)
36                            buffer2[i]=buffer1[i]-32;
37                    else
38                            buffer2[i]=buffer1[i];
39            }
40            printf("Your input is: %s \n",buffer2);
```

```
41     }
42     exit(0);
43  }
44
45  void intfunc(int signum)
46  {
47     printf("catch signal %d \n",signum);
48  }
49
50  void exitfunc(int signum)
51  {
52     printf("signal SIGTERM \n");
53     exit(0);
54  }
```

这个程序是由例1增加了2个信号处理(SIGINT和SIGTSTP)后得到的，程序的第12~26行分别注册了SIGINT、SIGTSTP和SIGTERM的信号处理函数intfunc和exitfunc，其中SIGINT、SIGTSTP都是由intfun函数来进行处理，intfunc函数的功能是输出捕获到的信号(第45~48行)。exitfunc函数的功能(第50~54行)同例1中的sigcatcher函数。主程序的其他部分(第27~43行)与例7-1相同。下面是程序的执行结果：

```
$ ./ex2
Pid of This Process : 6140
Please input:
Hello                          /*输入 Hello */
Your input is: HELLO

Please input:
catch signal 20                /*  按下 Ctrl+Z */
catch signal 2                 /*  按下 Ctrl+C */
signal SIGTERM                 /*在终端的另一窗口输入 kill 6140 ，程序退出。*/
```

6.2.2.2 高级信号处理

Linux系统还提供另一功能更强的系统调用sigaction：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

其中，参数signum指定要处理的信号(除SIGKILL和SIGSTOP之外)。act和oldact都是指向信号动作结构的指针。结构的定义如下：

```
struct sigaction
{
```

```
        void (*sa_handler)(int);
        void(*sa_sigaction)(int,siginfo_t *,void *);
        sigset_t sa_mask;
        int sa_flags;
}
```

其中 sa_handler 用于指向信号处理函数的地址。参数 sa_sigaction 是指向函数的指针。它指向的函数有三个参数，其中第二个为 siginfo_t 结构体，定义如下：

```
struct siginfo_t
{
        int si_signo;           /*Signal number */
        int si_errno;           /*An errno value */
        int si_code;      /*Signal code */
        pid_t si_pid;           /*Sending process ID */
        uid_t si_uid;           /*Real user ID of sending process */
        int si_status;          /*Exit value or signal */
        clock_t si_utime; /*User Time consumed */
        clock_t si_stime; /*System time consumed */
        signal_t si_value;/*Signal value */
        int si_int;             /* POSIX.1b signal */
        void *si_ptr;           /*POSIX.1b signal */
        void *si_addr;          /*Memory location that caused fault */
        int si_band;      /*Band event */
        int si_fd;              /*File descriptor */
}
```

sa_flags 指示信号处理函数的不同选项。具体可选参数见表 6-3。可以通过位运算的或运算(OR)串接不同的参数而实现所需的选项设置。将其赋值为 0 则选用所有的默认选项。

表 6-3　sa_flags 可选标志及对应设置

sa_flags	对 应 设 置
SA_NOCLDSTOP	用于指定信号 SIGCHLD，当子进程被中断时，不产生此信号，当且仅当子进程结束时产生该信号
SA_NOCLDWAIT	当信号为 SIGCHLD 时，此选项可以避免子进程的僵死
SA_NODEFER	当信号处理程序正在运行时，不阻塞对于信号处理函数自身的信号功能
SA_NOMASK	同 SA_NODEFER
SA_ONESHOT	当用户注册的信号处理函数被调用过一次之后，该信号的处理程序恢复为默认的处理函数
SA_RESETHAND	同 SA_ONESHOT
SA_RESTART	使本来不能进行自动重新运行的系统调用自动重新启动
SA_SIGINFO	表明信号处理函数是由 sa_sigaction 指定，而不是由 sa_handler 指定。它将显示更多处理函数的信息

函数 sigaction 不但可以实现函数 signal 的功能，而且还可以提供更加详细的信息，确切了解进程接收到信号时所发生的具体细节。它可以完全代替函数 signal 的功能。下面通过一个实例来说明这一点。

例 6-3 sigaction 函数的使用。

```
1  /* ex3.c */
2  #include <stdio.h>
3  #include <signal.h>
4  #include <string.h>
5  
6  void sighandler(int signum);
7  
8  int main()
9  {
10   char buffer1[100], buffer2[100];
11   int i;
12   struct sigaction act;
13   act.sa_handler=sighandler;
14   sigemptyset(&act.sa_mask);
15   act.sa_flags=0;
16   if(sigaction(SIGTERM, &act, NULL)==-1)
17   {
18           printf("Couldn't register signal handler!\n");
19           return 1;
20   }
21   printf("Pid of thi process: %d \n",getpid());
22  
23   for(;;)
24   {
25           printf("Please input:\n");
26           fgets(buffer1, sizeof(buffer1),stdin);
27           for(i=0;i<100;i++)
28           {
29                   if(buffer1[i]>=97&&buffer1[i]<=122)
30                           buffer2[i]=buffer1[i]-32;
31                   else
32                           buffer2[i]=buffer1[i];
33           }
34           printf("Your input is: %s \n",buffer2);
35   }
36   exit(0);
37 }
38 
```

```
39 void sighandler(int signum)
40 {
41   printf("catch signal SIGTERM. \n");
42   exit(0);
43 }
```

此程序与例 6-1 的功能完全一样，只是调用了不同函数实现对信号处理函数的注册，在此不再详细讲述。

6.2.3 信号集

在实际应用中，一个用户进程常常需要对多个信号作出处理。为了方便对多信号进行处理，在 Linux 系统中引入信号集的概念。这一节介绍与信号集相关的系统函数。

6.2.3.1 信号集的概念

信号集是用来表示多个信号的数据类型。通常可以用信号集和 sigpromask 一类的函数来通知内核允许信号集内的信号发生。POSIX.1 定义了数据类型 sigset_t 来表示信号集。可调用如下系统调用设定信号集中所包含信号：

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

其中参数 set 是指向信号集的指针。参数 signum 表示一个信号，可以用信号名表示。

函数 sigemptyset 将 set 所指向的信号集初始化为空，即不包括任何信号在内。函数 sigfillset 将 set 所指向的信号集初始化为包括所有的信号。所有的程序都要在每一个信号集使用之前，调用一次 sigemptyset 函数或 sigfillset 函数来对信号集进行初始化。这是因为我们不能确定 C 语言初始化的外部变量和静态变量，是否和给定系统的信号集的实现相适应。一旦初始化好一个信号集，就可以在信号集中添加信号和删除信号了。函数 sigaddset 的功能就是在指定的信号集中增加信号，函数 sigdelset 的功能就是在指定的信号集中去除一个信号。可以看到，上面所有的函数都是以信号集的指针为参数的。

上述四个函数调用成功时，返回值为 0；调用失败时，返回值为-1。

也可以通过函数调用查看某个特定信号是否属于一个信号集，这一操作的系统调用为：

```
#include <signal.h>
int sigismember(const sigset_t *set, int signum);
```

此函数用于检测一个信号 signum 是否在 set 所指向的信号集中。若 signum 属于该信

号集则返回值为1，不属于则返回值为0。

在后面的学习中将经常使用到信号集。

6.2.3.2 信号集的操作

1. sigprocmask 函数

进程的信号掩码(signal mask)是指传送给当前进程时被阻塞的信号的集合。进程可以通过调用以下函数来检测或改变信号掩码。

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

首先，如果 oset 是一个非空的指针，那么 oset 将返回进程当前的信号掩码(signal mask)；其次，如果 set 是一个非空的指针，那么参数 how 指出当前的信号掩码如何改变。表 6-4 给出了 how 的几种取值。其中，SIG_BLOCK 是一种或操作，而 SIG_SETMASK 是一种赋值操作。

上述函数执行成功返回 0，否则返回-1。

表 6-4 how 的取值含义

how	含　义
SIG_BLOCK	新的信号掩码是当前信号掩码和 set 所指向的信号集的和，即 set 所指向的信号集包含了我们要增加的被阻塞的信号
SIG_UNBLOCK	新的信号掩码是当前信号掩码和 set 所指向的信号集的差，即 set 所指向的信号集包含了我们要增加的不被阻塞的信号
SIG_SETMASK	进程的新信号掩码为 set 所指的信号集

如果 set 为一个空指针，那么进程的信号掩码将不会改变，这时 how 的位也是没有意义的。

下面例子程序的功能是打印调用进程的信号掩码中的信号的名称。

例 6-4 为进程打印信号掩码的例子。

```
/* ex4.c*/
#include <signal.h>

 void pr_mask(const char *str)
{
 sigset_t sigset;
 if(sigpromask(0, NULL, &sigset)<0)
 {
        printf("sigprocmask error");
        exit(0);
```

```
    }
    printf("%s", str);
    if(sigismember(&sigset, SIGINT))
            printf("SIGINT \n");
    if(sigismember(&sigset,SIGQUIT))
            printf("SIGQUIT \n");
    if(sigismember(&sigset, SIGUSR1))
            printf("SIGUSR1 \n");
    if(sigismember(&sigset, SIGALRM))
            printf("SIGALARM \n");
}
```

2. sigpending 函数

sigpending 函数返回在送往进程的时候被阻塞挂起的信号的集合。这个信号集通过参数 set 返回。它的调用格式如下：

```
#include <signal.h>
int sigpending(sigset_t *set);
```

sigpending 函数成功调用返回 0，否则返回-1。

例 6-5 信号集操作函数的使用。

```
1   /*ex5.c*/
2   #include <signal.h>
3   #include <stdio.h>
4
5   static void sig_quit(int);
6
7   int main()
8   {
9     sigset_t newmask, oldmask, pendmask;
10    if(signal(SIGQUIT,&sig_quit)==-1)
11    {
12            printf("Couldn't register signal hanlder for SIGQUIT!\n");
13            exit(1);
14    }
15    sigemptyset(&newmask);
16    sigaddset(&newmask,SIGQUIT);
17
18    if(sigprocmask(SIG_BLOCK, &newmask, &oldmask)<0)
19    {
20            printf("SIG_BLOCK error.\n");
21            exit(2);
22    }
```

```
23      sleep(5);
24      if(sigpending(&pendmask)<0)
25      {
26              printf("sigpending    error.\n");
27              exit(3);
28      }
29      if(sigismember(&pendmask,SIGQUIT))
30              printf("SIGQUIT pending \n");
31      if(sigprocmask(SIG_SETMASK, &oldmask, NULL)<0)
32      {
33              printf("SIG_SETMASK    error.\n");
34              exit(4);
35      }
36      printf("SIGQUIT unblocked.\n");
37      sleep(5);
38      exit(0);
39  }
40
41  static void sig_quit(int signum)
42  {
43      printf("catch SIGQUIT.\n");
44      if(signal(SIGQUIT, SIG_DFL)==-1)
45              printf("Couldn't reset SIGQUIT!\n");
46      return;
47  }
```

分析：以上程序中，首先用 signal 函数注册了 SIGQUIT 信号处理函数，阻塞了 SIGQUIT 信号(第 10~14 行)，接着用 sigemptyset 函数将信号集初始化为空(第 15 行)，然后把 SIGQUIT 信号添加到信号集中(第 16 行)。并且保存了当前的信号掩码(signalmask)(为了返回时恢复)(第 18~22 行)，然后睡眠 5 秒钟(第 23 行)。在此期间发生的 quit 信号都将被阻塞挂起，而且在信号不被阻塞之前都不会送给进程。在进程睡眠 5 秒钟之后，会检测是否有挂起的信号(第 24~28 行)，然后唤醒这些被挂起的信号(第 29~30 行)。为了消除对信号的阻塞，对原有的掩码做 SIG_SETMASK 操作，并不对阻塞的信号做 SIG_UNBLOCK 操作(第 31~35 行)。如果写一个被其他程序调用的过程，并且在函数中阻塞某个信号，就不能使用 SIG_UNBLOCK 来给信号解除阻塞。在前面的程序中使用了 SIG_SETMASK 来将信号掩码恢复为原有值，这是因为程序在调用这个函数之前可能阻塞了这个信号。

进程在第二次调用 sleep 函数之后会睡眠 5 秒钟(第 37 行)，如果在进程睡眠期间产生 quit 信号，信号会结束进程的运行，因为我们在前面接收信号之后将信号处理函数设为缺省函数。

程序运行结果如下：

```
$ ./ex5                /* 运行程序 */
                       /* 进程睡眠未到 5 秒钟时，输入“Ctrl+\”，产生 SIGQUIT 信号。*/
SIGQUIT pending        /*从 sleep 函数返回后 */
catch SIGQUIT.         /* 在信号处理函数中 */
SIGQUIT unblocked.     /* 从 sigprocmask 函数中返回 */
退出 (core dumped)     /*再次输入“Ctrl+\”，产生 SIGQUIT 信号。*/
```

Shell 在看到子进程非正常结束时就会打印“退出 (core dumped)”信息。

3. sigsuspend 函数

sigsuspend 函数用于使进程挂起。它的调用格式如下：

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

参数 sigmask 指向一个信号集，当函数 sigsuspend 被调用时，sigmask 所指向的信号集中的信号被赋值给信号掩码。之后进程被挂起，直至进程捕捉到信号调用处理函数并执行完毕返回时，函数 sigsuspend 返回。信号掩码恢复为函数调用前的值。进程结束信号可以使其立刻终止。

下面来看一个例子。

例 6-6 使用 sigsuspend 函数使进程挂起等待某个全程变量的例子。

```
1   /* ex6.c */
2   #include <signal.h>
3   #include <stdio.h>
4
5
6   int quitflag=0;
7
8   int main()
9   {
10    void sig_int(int);
11    sigset_t newmask, oldmask, zeromask;
12
13    if(signal(SIGINT,&sig_int)==-1)
14    {
15            printf("Couldn't register signal hanlder for SIGINT!\n");
16            exit(1);
17    }
18    if(signal(SIGQUIT,&sig_int)==-1)
19    {
20            printf("Couldn't register signal hanlder for SIGQUIT!\n");
21            exit(2);
```

```
22   }
23   sigemptyset(&zeromask);
24   sigemptyset(&newmask);
25   sigaddset(&newmask,SIGQUIT);
26   if(sigprocmask(SIG_BLOCK, &newmask, &oldmask)<0)
27   {
28           printf("SIG_BLOCK   error.\n");
29           exit(3);
30   }
31   while(quitflag==0)
32           sigsuspend(&zeromask);
33   printf("process wake up.\n");
34   quitflag=0;
35   if(sigprocmask(SIG_SETMASK,&oldmask,NULL)<0)
36           exit(0);
37 }
38
39 void sig_int(int signum)
40 {
41   if(signum==SIGINT)
42           printf("\n interrupt \n");
43   else if (signum==SIGQUIT)
44   {
45           printf("catch SIGQUIT \n");
46           quitflag=1;
47   }
48   return ;
49 }
```

分析：以上程序中，首先用 signal 函数注册了 SIGINT 和 SIGQUIT 信号处理函数(第 13~22 行)，接着用 sigemptyset 函数将信号集初始化为空(第 23 行)，然后把 SIGQUIT 信号添加到信号集中(第 25 行)。并且保存了当前的信号掩码(signalmask)(为了返回时恢复)(第 26~30 行)，然后程序用 sigsuspend 函数将进程挂起(第 31~32 行)，直到全局变量 quitflag 由 0 变为 1 为止，这是由信号处理函数 sig_int 捕获到 SIGQUIT 后进行改变的(第 43~47 行)。

程序运行结果如下：

```
$ ./ex6
catch SIGQUIT          /*进程一直处于挂起状态，输入“Ctrl+\”，产生 SIGQUIT 信号*/
process wake up.       /*  进程从挂起状态恢复  */
```

6.2.4 发送信号

和信号的处理比较，信号的发送要简单得多。信号发送的关键是使系统知道向哪个进程发送信号以及发送什么信号。只要明确了这两点，就可以很方便地发送信号了。但需要注意的是，能否向某一进程发送某一特定信号是和用户的权限密切相关的。例如，只有系统管理员才能向任何一个进程发送 SIGKILL 信号终止该进程。

先来看几个用于信号发送的系统调用。

1. raise 函数

raise 函数用于向一个进程自身发送信号。它的调用格式如下：

```
#include <signal.h>
int raise(int signum);
```

参数 signum 为所发送的信号编号。调用成功时，返回值为 0，调用失败返回值为-1。

例 6-7 raise 函数的使用。

```
1   /* ex7.c */
2   #include <stdio.h>
3   #include <signal.h>
4
5   void inthandler(int signum);
6   void continuehandler(int signum);
7   void terminatehandler(int signum);
8
9   int main()
10  {
11    char buffer[100];
12    if(signal(SIGINT,&inthandler)==-1)
13    {
14          printf("Couldn't register signal hanlder for SIGINT!\n");
15          exit(1);
16    }
17    if(signal(SIGTSTP, &inthandler)==-1)
18    {
19          printf("Couldn't register signal hanlder for SIGTSTP!\n");
20          exit(2);
21    }
22    if(signal(SIGCONT, &continuehandler)==-1)
23    {
24          printf("Couldn't register signal hanlder for SIGCONT!\n");
25          exit(3);
```

```
26    }
27    if(signal(SIGTERM, &terminatehandler)==-1)
28    {
29            printf("Couldn't register signal hanlder for SIGINT!\n");
30            exit(4);
31    }
32    printf("Pid of This Process : %d \n",getpid());
33
34    for(;;)
35    {
36            printf("Please input:\n");
37            fgets(buffer, sizeof(buffer),stdin);
38            if(strcmp(buffer,"int\n")==0)
39                    raise(SIGINT);
40            else if(strcmp(buffer,"stop\n")==0)
41                    raise(SIGTSTP);
42            else if(strcmp(buffer,"continue\n")==0)
43                    raise(SIGCONT);
44            else if(strcmp(buffer,"quit\n")==0)
45                    raise(SIGTERM);
46            else
47                    printf("Your input is: %s \n",buffer);
48    }
49    exit(0);
50  }
51
52  void inthandler(int signum)
53  {
54    printf("catch signal %d \n",signum);
55  }
56
57  void continuehandler(int signum)
58  {
59    printf("Continue code.\n");
60  }
61
62  void terminatehandler(int signum)
63  {
64    printf("signal SIGTERM \n");
65    exit(0);
66  }
```

分析：此程序能处理的信号同程序 2。不同的是，当由终端输入某几个特定的字符串

int、stop、continue 和 quit 时，程序不再将字符串处理并回显，而是向其自身发送信号，再调用相应的信号处理函数(第 38~45 行)。程序的执行结果如下：

```
$ ./ex7
Pid of This Process : 8669
Please input:                          /* 程序等待输入 */
hello                          /* 输入 hello */
Your input is: hello

Please input:
int                            /* 输入 int */
catch signal 2
Please input:
stop                           /* 输入 stop */
catch signal 20
Please input:
continue                       /* 输入 continue */
Continue code.
Please input:
quit                           /* 输入 quit */
signal SIGTERM
```

2. kill 函数

kill 函数发信号给一个进程或一组进程。它的调用格式如下：

```
#include <signal.h>
int kill(pid_t pid, int signum);
```

参数 pid 参数表示 kill 函数发送信号对象的进程号或进程组号。具体对应关系见表 6-5 所示。signum 指明要发送的信号。

表 6-5　参数 pid 的取值含义

取　　值	含　　义
pid>0	向进程号为 pid 值的进程发送信号
pid=0	向与发送信号的进程有相同进程组号的进程发送信号
pid<-1	向进程组号为 pid 绝对值的进程组发送信号
pid=-1	未定义

当信号成功发送时，kill 函数返回 0；发生错误时，返回-1。

用 kill 向进程或进程组发送信号时还有一些许可权限制，这些限制用于防止恶意的行为，如随意杀掉属于其他用户的进程。进程是否有权发送信号给另一个进程是由这两个进程的用户 ID 决定的。一般情况下，发送信号进程的实际用户 ID 或有效用户 ID 必须与接

收信号进程的实际用户 ID 或有效用户 ID 相同。超级用户可以向任何进程发送信号。例 6-8 给出了 kill 函数的使用。

例 6-8 kill 函数的使用。

```
1    /* ex8.c*/
2    #include <signal.h>
3    #include <stdio.h>
4    #include <stdlib.h>
5    #include <unistd.h>
6
7    int usr_interrupt=0;
8
9    void synch_signal(int signum)
10   {
11     usr_interrupt=1;
12   }
13
14   void child_function()
15   {
16     printf(" I'm child process. My pid is %d \n",getpid());
17     sleep(5);
18     kill(getppid(),SIGUSR1);
19     puts("Good bye! \n");
20     exit(0);
21   }
22
23   int main()
24   {
25     struct sigaction usr_action;
26     sigset_t block_mask;
27     pid_t child_id;
28
29     sigfillset(&block_mask);
30     usr_action.sa_handler=synch_signal;
31     usr_action.sa_mask=block_mask;
32     usr_action.sa_flags=0;
33     sigaction(SIGUSR1, &usr_action,NULL);
34     child_id=fork();
35     if(child_id==0)
36             child_function();
37     while(!usr_interrupt);
38     puts("That's all!");
39     return 0;
```

```
40    }
```

分析：上述程序说明怎样将信号用于进程间的通信。在这个例子中，定义了一个全局变量，并初始化为 0，在信号处理函数中将改变为 1(第 7~12 行)。第 14~21 行是子进程函数，子进程首先输出自己的进程号，然后休眠 5 秒钟，之后用 kill 函数向父进程发送一 SICUSR1 信号通知父进程，退出。在 main 函数中，父进程派生一子进程(第 34 行)，然后等待子进程发送信号(第 37 行)。收到信号后，父进程继续执行，输出相关信息后退出(第 38~40 行)。程序执行结果如下：

```
$ ./ex8
I'm child process. My pid is 9117          /* 等待子进程从休眠中恢复 */
Good bye!                                   /* 子进程发送信号后退出 */

That's all!                                 /* 主进程收到信号后退出 */
```

6.3 管　　道

本节介绍管道通信机制的基本概念，创建管道的两种方法，以及管道使用的一般方法。

6.3.1 基本概念

在介绍管道的使用方法之前，先来介绍一下管道的相关概念，如管道、管道的内部实现、管道的读写操作和管道的局限性等。

6.3.1.1 管道的概念

管道，就是将一个进程的标准输出和另一个进程的标准输入联系到一起，以供两个进程相互通信的方法。管道是 UNIX 中最古老的进程通信机制，Linux 中也提供了管道。它的应用非常广泛，就连 Linux 命令行中也有使用，如：

```
$ls | sort | head -5
```

这条命令中，ls 的输出作为 sort 的输入，sort 的输出又作为 head -5 的输入，head -5 的输出将出现在屏幕上。这条命令的最终执行结果是将文件列表排序，但只输出前五行。从中可以直观地看到管道的特点。

6.3.1.2 管道的内部实现

那么，管道的内部实现是怎样的呢？当一个进程创建一个管道时，系统内核为使用管

道准备了两个文件描述符。一个用于管道的输入，也就是在管道中写入数据；另一个用于管道的输出，也就是从管道中读出数据。这样，用户程序的系统调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作。

如果一个管道只与一个进程相联系，只实现进程自身内部的通信，它就是毫无用途的。所以一般来说，创建管道的进程接着就会创建其子进程，由于父子进程可以共享打开文件，子进程会从父进程那里继承到读写管道的文件描述符，这样，父子进程间的通信管道就建立起来了。

接下来，要确定数据的传输方向，是从子进程传送到父进程，还是从父进程传送到子进程。这一点确定之后，父子进程分别关闭与之无关的那个描述符。比如数据从子进程传送到父进程，则子进程关闭读管道的描述符，父进程关闭写管道的描述符。这样，就建立了从子进程到父进程的通信管道。

6.3.1.3　管道的读写操作

通信管道建立以后，可以通过调用 read 和 write 函数来读写管道，完成信息的传递。由于管道的一端已经关闭，读写管道应注意下面的情况：

- 如果从一个写描述符关闭的管道中读数据，当读完所有的数据后，read 函数返回 0，表明已到达文件末尾。其实，严格来说，只有当没有数据继续写入后，才可以说到达了文件末尾。所以应该分清到底是暂时没有数据输入，还是已经到达文件末尾，如果是前者，读进程应该等待。多进程写，单进程读的情况就更加复杂。
- 如果向一个读描述符关闭的管道中写数据，就会产生 SIGPIPE 信号。不管是忽略这个信号，还是处理它，write 函数都将会返回-1。
- 常数 PIPE_BUF 规定了内核中管道缓冲的大小，所以在写管道时要注意这一点。一次向管道中写入 PIPE_BUF 或更少的字符，不会和其他进程写入的内容交错；反之，当存在多个写管道的进程时，向其中写入超过 PIPE_BUF 个字符时，就会产生交错现象。

6.3.2　管道的创建

创建管道的系统调用如下所示：

```
#include <unistd.h>
int pipe(int fdes[2]);
```

其中，参数 fdes 为整数数组名，在 C 语言中，数组名即是指向数组的指针。所以，调用这个函数后，系统为通道分配的两个文件描述符将通过这个数组返回到用户进程中。fdes 中的第一个整数是为读通道准备的，第二个整数是为写通道准备的。可以说，fdes[1]的输出是 fdes[0]的输入。

当调用成功时，pipe 返回 0，否则返回-1。

管道有以下一些特点：

- 管道没有名字，它是为了一次使用而创建的。
- 管道的两个描述符是同时打开的。如果从一个没有任何进程向它写的管道读数据(由于这些进程已关闭了所有文件或已退出)，read 将返回文件结束。如果往一个没有任何进程读它的管道写数据，则视为错误；这将导致生成 SIGPIPE 信号，并且当信号被阻塞时以 EPIPE 错误失败。
- 管道不允许文件定位。读和写操作都是顺序的，读从文件的开始处读，写则写至文件尾。

下面来看一个简单的例子。

例 6-9　使用 pipe 创建管道的例子。

```
1   /* ex9.c */
2   #include <unistd.h>
3   #include <stdio.h>
4   #include <sys/types.h>
5
6   int main()
7   {
8     int n, fd[2];
9     pid_t pid;
10    char buffer[BUFSIZ+1];
11    if(pipe(fd)<0)
12    {
13            printf("pipe failed!\n ");
14            exit(1);
15    }
16    if((pid=fork())<0)
17    {
18            printf("fork failed!\n ");
19            exit(1);
20    }
21    else if (pid>0)
22    {
23            close(fd[0]);
24            write(fd[1],"How are you?\n",12);
25    }
26    else
27    {
28            close(fd[1]);
29            n=read(fd[0],buffer,BUFSIZ);
30            write(STDOUT_FILENO,buffer,n);
31    }
```

```
32    exit(0);
33  }
```

分析：这是一个父进程创建管道，并通过它将数据传递给子进程的例子。整个程序非常简单。父进程首先创建一个管道(第 11~15 行)，然后通过 fork 语句创建子进程(第 16~20 行)，关闭管道的读功能(第 23 行)，向管道中写入数据(第 24 行)；子进程关闭管道的写功能(第 28 行)，从管道中读出数据(第 29 行)，并输出到标准输出中 (第 30 行)。程序的执行结果如下：

```
$ ./ex9
$ How are you?
```

上面的例子只是简单地使用了管道的文件描述符。更常见的情况是子进程调用 dup 或 dup2 函数，将管道的文件描述符复制到标准输入或输出上，接着子进程调用 exec 函数运行其他程序，那么这个程序的标准输入或标准输出就成为从管道读入或向管道输出了。下面看一个简单的例子。

例 6-10 假设有一个用户可执行程序 upcase，可以从标准输入设备读入字母，将其从小写转化为大写输出。编写一个程序用管道实现将某一文本文件中的字母转化为大写输出的程序。其中，文本文件名作为参数传进来。

```
1   /* ex10.c */
2   #include <unistd.h>
3   #include <stdio.h>
4   #include <sys/types.h>
5   #include <sys/wait.h>
6
7   int main(int argc, char *argv[])
8   {
9     int n,fd[2];
10    pid_t pid;
11    char buffer[BUFSIZ+1];
12    FILE *fp;
13
14    if(argc<=1)
15    {
16            printf("usage: %s <pathname>\n",argv[0]);
17            exit(1);
18    }
19    /*  打开文本文件  */
20    if((fp=fopen(argv[1],"r"))==NULL)
21    {
22            printf("Can't open %s \n", argv[1]);
```

```
23          exit(1);
24     }
25     /*  创建管道  */
26     if(pipe(fd)<0)
27     {
28          printf("pipe failed!\n ");
29          exit(1);
30     }
31     /*  创建子进程  */
32     if((pid=fork())<0)
33     {
34          printf("fork failed!\n ");
35          exit(1);
36     }
37     else if (pid>0)                         /*  父进程  */
38     {
39          close(fd[0]);
40          while(fgets(buffer,BUFSIZ,fp)!=NULL)
41          {
42               n=strlen(buffer);
43                /*  向管道中写入数据  */
44               if(write(fd[1],buffer,n)!=n)
45               {
46                    printf("write error to pipe.\n");
47                    exit(1);
48               }
49          }
50          if(ferror(fp))
51          {
52               printf("fgets error. \n");
53               exit(1);
54          }
55          close(fd[1]);
56          if(waitpid(pid, NULL, 0)<0)
57          {
58               printf("waitpid error!\n");
59               exit(1);
60          }
61          exit(0);
62     }
63     else                                    /*  子进程  */
64     {
65          close(fd[1]);
```

```
66          if(fd[0]!=STDIN_FILENO)
67          {
68              /* 将管道复制到标准输入 */
69              if(dup2(fd[0],STDIN_FILENO)!=STDIN_FILENO)
70              {
71                  printf("dup2 error to stdin! \n");
72                  exit(1);
73              }
74              close(fd[0]);
75          }
76          /* 运行用户程序 */
77          if(execl("upcase","upcase",(char *)0)<0)
78          {
79              printf("execl error for upcase.\n");
80              exit(1);
81          }
82          exit(0);
83      }
84  }
```

说明：在这个程序中，首先创建管道文件(第 26~30 行)，然后调用了 fork 函数(第 32~36 行)。fork 函数之后，父进程关闭管道的读功能(第 39 行)，而子进程关闭管道的写功能。(第 65 行)接着，父进程开始从文本文件中读出数据，写入管道中(第 40~49 行)；子进程调用 dup2 将它的标准输入设定为管道的读数据的一端(第 69~73 行)，这样，当 upcase 程序运行时，它的标准输入就是管道的读数据端了，也就是父进程写入的数据，即那个文本文件。之后运行 upcase 程序完成文本转换(第 77 ~81 行)。

程序运行结果如下：

```
$ cat text              /* 显示文本文件 text 的内容 */
how are you?
$ ./ex10 text           /*以 text 文本文件作为输入参数运行程序 */
HOW ARE YOU?
```

进程之间常常有竞争产生，所以应该采取些措施实现进程的同步控制。下面就是一个使用管道达到进程同步控制的例子。

6.3.3 创建管道的简单方法

前面一节介绍了创建管道的方法，可以看到两个过程使用管道通信的操作对用户来说是比较繁杂的。为此，系统提供了创建管道的简单方法：popen 和 pclose 函数。如果使用这两个函数创建管道，所有复杂的操作都在内部完成了，如创建管道，fork 子进程，关闭

管道的无用端，执行一个 shell 命令，等待命令执行等。这两个函数的描述如下：

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);
int pclose(FILE *fp);
```

这两个函数的作用类似于 fopen 和 fclose，具体说明如下：

(1) 函数 popen 用于创建管道。它内部调用 fork 和 exec 函数执行命令行 cmdstring，返回一个 FILE 结构的指针，即用于访问管道的指针。

(2) popen 中的参数 const char *cmdstring 就是一个命令行。所有的 shell 命令行参数和选项都可以使用。如下面的函数调用都是合法的：

```
popen("ls *.*","r");
popen("sort > /tmp/foo","w");
popen("sotr | uniq | more", "w");
```

(3) popen 中的参数 const char *type 指出管道的类型。如果管道是以类型"r"打开的，那么这个管道的输入端连接到了命令行 cmdstring 的标准输出端。此时，命令行的输出可以从管道中读入。反之，如果管道是以类型"w"打开的，那么这个管道的输出端连接到了命令行的标准输入端。此时，向管道中写入的数据就成为命令行的输入数据。可以看到，type 的作用与 fopen 和 fclose 中的相同，可以取"r"或"w"，表示管道可读或可写，但决不可以即可读又可写。在 Linux 系统下，规定管道的打开方式取决于 type 的第一个字符，比如 type 为"rw"，那么管道就是以"r"方式，即可读方式打开的。

(4) 函数 pclose 是用来关闭管道的。它关闭标准输入输出流，等待命令行执行完毕，然后返回结束时的状态。如果 shell 不能执行这个命令行，结束时的状态就如同在 shell 中执行了 exit(127)。

下面来看一个使用简易方法创建管道的例子。

例 6-11 用创建管道的简易方法实现例 6-10 的程序。

```
1    /* ex11.c */
2
3    #include <stdio.h>
4    #include <sys/types.h>
5    #include <sys/wait.h>
6
7    int main(int argc, char *argv[])
8    {
9      char buffer[BUFSIZ+1];
10     FILE *fpin, *fpout;
11     if(argc<=1)
12     {
13            printf("usage: %s <pathname>\n",argv[0]);
```

```
14              exit(1);
15          }
16          if((fpin=fopen(argv[1],"r"))==NULL)
17          {
18              printf("Can't open %s \n", argv[1]);
19              exit(1);
20          }
21          if((fpout=popen("upcase","w"))==NULL)
22          {
23              printf("popen error \n");
24              exit(1);
25          }
26          while(fgets(buffer,BUFSIZ,fpin)!=NULL)
27          {
28              if(fputs(buffer,fpout)==EOF)
29              {
30                  printf("fputs error to pipe. \n");
31                  exit(1);
32              }
33          }
34          if(ferror(fpin))
35          {
36              printf("fgets error. \n");
37              exit(1);
38          }
39          if(pclose(fpout)==-1)
40          {
41              printf("pclose error.\n");
42              exit(1);
43          }
44          exit(0);
45      }
```

说明这段程序十分简单。首先打开文本文件(第 16~20 行)，通过函数 popen 创建一个可写的管道，将命令行 upcase 的输入与管道的输出连接在一起(第 21~25 行)，然后向管道中输入数据(第 28~32 行)，那么命令行就可以通过管道接收到文本文件的数据了。

6.3.4 命名管道

前面一节介绍了管道的有关操作，本节介绍进程通信的另一种机制：命名管道。这里详细地介绍它的概念、创建操作和应用。

6.3.4.1 命名管道的特点

命名管道又叫先进先出队列，它类似于前面介绍的管道，但又有些自己的特点：

- 前面介绍的管道只能用于相同祖先的进程间通信，而命名管道就没有这个限制，没有任何联系的进程间也可以使用这种机制进行通信。
- 前面介绍的管道是在内核中存储的，程序运行结束后将不复存在，而命名管道是作为特殊设备文件存储在文件系统中的，当程序运行结束后，它继续存在于文件系统中备用。除非删除该文件，否则这个命名管道不会消失。

在 shell 环境下，可以很简单地识别出命名管道文件。文件名后面紧跟着一个竖线，就是命名管道文件的标志。而在程序中，由于命名管道文件是一种特殊类型的文件，可以通过 S_ISFIFO 宏来检测。

6.3.4.2 命名管道的操作

一旦命名管道建立，就可以像普通文件那样，对其使用 open、close、read、write、unlink 等文件操作了。请注意，由于命名管道是个特殊的文件，不像普通管道那样存在于内核中，仅仅创建并不能立即使用，必须打开才能进行读写操作。读写操作要特别注意以下几点：

(1) 像普通管道那样，如果没有其他写进程打开一个命名管道就对其进行读操作，会产生 SIGPIPE 信号；如果所有的写进程都关闭命名管道，对其的读操作就会认为到达文件末尾。

(2) 在多个写进程的情况下，写交错现象就有可能发生。与普通管道相同，只要一次写入的字符数不超过 PIPE_BUF，就不会产生写交错现象。

命名管道常常产生阻塞状态。也就是说，如果一个读进程打开命名管道，那么这个进程就要进入阻塞状态，直到其他写进程打开这个管道为止。同样，如果一个写进程打开命名管道，这个进程也会出现阻塞状态，直到其他读进程打开这个管道为止。

如果用户不希望出现这种阻塞状态，可以通过设置 O_NONBLOCK 标志来实现。这样，不管有没有写进程，读打开操作就会立即返回。但是，如果没有读进程，写打开操作就会产生错误。

6.3.4.3 命名管道的应用

命名管道用在 2 个方面：

(1) shell 命令行使用命名管道将数据从一条命令传到另一条命令，而不需创建中间的临时文件。

(2) 在客户——服务器结构中，使用命名管道在客户和服务器之间交换数据。

6.3.4.4 命名管道的创建

命名管道可以在程序中通过函数调用生成。由于命名管道生成后不会自动消失，所以一般不在程序中生成。但在有些特殊情况下还是需要的。

创建命名管道文件的函数有 2 个，形式如下所示：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
int mknod(char *pathname, mode_t mode, dev_t dev);
```

这 2 个函数中，mknod 不是专门用来创建命名管道文件的，而 mkfifo 是。其中，参数 mode 的取值要求与前面介绍文件系统中的 open 函数的 mode 的取值相同，而参数 dev_t dev 只有在文件为设备文件时才起作用。

这 2 个函数调用成功时返回 0，否则返回-1。

我们应用命名管道实现一个简单的客户－服务器模式。在单服务器多客户的模式下，服务器打开一个所有客户进程都可以访问的命名管道，通过这个管道接受客户进程的信息。客户进程的每次写操作都不要超过 PIPE_BUF 个字符，这样可以避免写交错现象。程序代码如例 6-12 所示。

例 6-12　编写一个多客户－单一服务器模式的程序，用命名管道实现客户到服务器之间传递数据的操作。

首先实现服务器程序。

```
1   /* ex12 server.c */
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <sys/stat.h>
5   #include <unistd.h>
6   #include <linux/stat.h>
7
8   #define FIFO_FILE "MYFIFO"
9
10  int main()
11  {
12    FILE *fp;
13    char readbuf[80];
14    /*  如果命名管道文件不存在，要先创建一个。  */
15    if((fp=fopen(FIFO_FILE,"r"))==NULL)
16    {
17            umask(0);
18            mknod(FIFO_FILE,S_IFIFO|0666,0);
19    }
20    else
21            fclose(fp);
22    while(1)
23    {
24          /*  打开命名管道文件  */
25            if((fp=fopen(FIFO_FILE,"r"))==NULL)
```

```
26          {
27                  printf("open fifo failed. \n");
28                  exit(1);
29          }
30      /*  从命名管道中读数据  */
31          if(fgets(readbuf,80,fp)!=NULL)
32          {
33                  printf("Received string :%s \n", readbuf);
34                  fclose(fp);
35          }
36          else
37          {
38                  if(ferror(fp))
39                  {
40                          printf("read fifo failed.\n");
41                          exit(1);
42                  }
43          }
44    }
45    return 0;
46  }
```

说明：服务器程序如果不出错，基本上一直运行下去，等待接受客户进程的输入。它首先创建一个命名管道(第 15~21 行)，然后不停重复下面的操作：打开(第 25~29 行)，接受一行信息，输出在屏幕上，关闭(第 31~35 行)。

接下来实现客户程序。

```
1   /* ex12 client.c */
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   #define FIFO_FILE "MYFIFO"
6
7   int main(int argc, char *argv[])
8   {
9     FILE *fp;
10    int i;
11    if(argc<=1)
12    {
13            printf("usage: %s <pathname>\n",argv[0]);
14            exit(1);
15    }
16    if((fp=fopen(FIFO_FILE,"w"))==NULL)
```

```
17    {
18          printf("open fifo failed. \n");
19          exit(1);
20    }
21    for(i=1;i<argc;i++)
22    {
23          if(fputs(argv[i],fp)==EOF)
24          {
25                printf("write fifo error. \n");
26                exit(1);
27          }
28          if(fputs(" ",fp)==EOF)
29          {
30                printf("write fifo error. \n");
31                exit(1);
32          }
33    }
34    fclose(fp);
35    return 0;
36  }
```

说明：客户程序只运行一段时间就停止了。它打开命名管道(第 16~20 行)，向其中写入一行信息(第 21~33 行)，关闭命名管道(第 34 行)，退出运行态(第 35 行)。

程序运行结果如下：

首先在一个终端窗口中运行 server 程序，运行后，程序就停留在等待接收状态，如下所示：

```
$ ./server
```

然后在另一个终端窗口中运行 client 程序：

```
$./client how are you?
```

这时在 server 窗口中会显示：

```
Received string :how are you?
```

6.4 System V IPC 机制

为了和其他系统保持兼容，Linux 也提供了三种原先出现在 UNIX System v 中的 IPC

机制。这三种机制分别是：消息队列、信号量以及共享内存。这 3 种 IPC 机制在编程接口和内部实现上都非常类似。例如，它们都有一个唯一的整数标识，都有一个关键字，使用某种相同的数据结构，并且都有类似的获取和控制 IPC 资源的函数，如：msgget、semget 和 shmget、msgctl、semctl 和 shmctl。因此，在具体讲述每一种 IPC 机制之前，先介绍它们的共同之处。为了叙述方便，我们统称单个消息队列、信号量集合以及共享存储为 IPC 资源。

6.4.1 基本概念

本小节介绍三种 IPC 机制的一些基本概念，如引用标识符、关键字等，也介绍了某些数据结构，如 ipc_term 等。

6.4.1.1 关键字和标识符

每一个 IPC 资源有 2 个唯一的标志与之相连：关键字(key)和标识符(id)。

1. 标识符

每个 System V 的进程通信机制中的对象都和唯一的一个引用标识符相联系，如果进程要访问此 IPC 对象，则需要在系统中传递这个唯一的引用标识符。例如，要访问某个共享内存段，唯一需要的就是指定给这个内存段的标识符，只有通过这个标识符才可以完成相关的操作：

标识符的唯一局限在相应的 IPC 对象的类别内。为了说明这一点，假设“12345”是某个消息队列的标识符，那么肯定不会有第二个消息队列的标识符为“12345”，但是某个共享内存或者某个信号集的标识符却有可能是“12345”。

2. 关键字

关键字(key)是用来定位 System V 中 IPC 机制的对象的引用标识符的。当创建一个 IPC 机制的对象时，必须指定一个关键字。关键字的类型 key_t，是系统中预先规定的，它定义在头文件<sys/types.h>中。

关键字的取值可以为 IPC_PRIVATE 或者其他值不等于 IPC_PRIVATE 的整数。IPC_PRIVATE 定义在<sys/ipc.h>中，其值通常为 0。这个关键字有特殊的含义，它表示总是创建一个新的 IPC 资源。因为用这个关键字调用内核总是分配新的 IPC 资源，所以它实际上表示所创建的 IPC 资源是创建进程私有的，即其他进程不可能得到它。但是，创建这个 IPC 资源的进程可以与其子进程共享该资源，子进程通过 fork 继承父进程的 IPC 资源。

其他不等于 IPC_PRIVATE 的关键字可以是直接指定的整常数，例如 12345，也可以是由公共种子导出的值。例如，C 库中提供了一个函数 ftok 可以将文件名转换为关键字。这个函数的原型为：

```
#include <sys/ipc.h>
```

```
key_t ftok(const char *path, int id);
```

ftok 函数根据 path 和 id 返回一个类型为 key_t 的关键字。

path 参数必须是一个已存在文件的路径名。id 只有低 8 位有效。对于命名同一个文件的所有路径名，当用同样的 id 调用 ftok 函数，该函数返回相同的关键字；当用不同的 id 调用 ftok 函数时，返回不同的关键字。如果 id 低 8 位为 0，ftok 的行为不确定。

ftok 返回的关键字是根据文件的 inode 确定的，因此，如果这个文件在删除后又重新创建，则由 ftok 返回的关键字也会改变，尽管路径名仍然一样。

使用 IPC 资源通信的进程虽然可以直接用诸如 1234 这样的整数作为关键字，但它们之间需要在程序编码上保持一致，并且，这样做还有一个更致命的弱点：其他进程也可能使用这个整数作为另外的 IPC 资源的关键字。在这种情况下，则有可能导致混乱。因此，最好用 ftok 函数来生成 IPC 资源的关键字。

6.4.1.2 ipc_perm 结构介绍

每一个进程通信机制的对象即有一个 ipc_perm 结构与之对应，这个结构中记录了对象的一些信息，如所有者、创建者和权限等。它定义在头文件<sys/ipc.h>中，具体定义如下所示：

```
struct ipc_perm{
    uid_t uid;          /*所有者的有效用户 ID */
    gid_t gid;          /*所有者的有效组 ID */
    uid_t cuid;              /*创建者有效用户 ID */
    gid_t cgid;              /*创建者的有效组 ID */
    mode_t mode;        /*访问权限  */
    ulong seq;          /*应用序号  */
    key_t key;          /*关键字  */
};
```

下面详细说明这个结构的某些域的含义：

- uid、gid、cuid 和 cgid：这四个域中记录了 IPC 机制的对象的所有者和创建者的信息，它们是在创建对象时确定下来的，也可以通过系统函数的调用修改它们的值。但是，有权限修改这些值的只能是对象的创建者或超级用户。这与文件系统中的 chown 和 chmod 有些类似。
- mode：这个域记录了 IPC 机制的对象的访问权限，它与文件的访问权限有些类似，同样，用户、组用户和其他用户这三类不同用户的权限不同。但是，这些权限中没有可执行权限了，并且术语也有些改变。消息队列和共享内存的权限使用术语“可读”、“可写”，而信号量则使用术语“可读”和“可改变”。表 6-6 中记录了不同 IPC 机制分别对应的 6 种权限。
- seq：这个域记录了 IPC 机制的对象的应用序号，它并不是确定的值。每次对象被使用，这个值都会增加一，直到整数的最大值，然后又重新从 0 开始。用户不需要

对它有很深的了解。

- key：这个域记录了进程通信对象的关键字的值。

表 6-6　IPC 机制的权限

权　　限	消 息 队 列	信　号　量	共 享 内存
用户可读	MSG_R	SEM_R	SHM_R
用户可写	MSG_W		
组用户可读	MSG_R>>3	SEM_R>>3	SHM_R>>3
组用户可写	MSG_W>>3	SEM_A>>3	SHM_W>>3
其他用户可读	MSG_R>>6	SEM_R>>6	SHM_R>>6
其他用户可写	MSG_W>>6	SEM_A>>6	SHM_W>>6

6.4.1.3　ipcs 命令

可以使用 ipcs 命令得到 IPC 机制中所有对象的状态。这条命令可带选项，“-q”“-s”“-m”分别表示只输出消息队列、信号量、共享内存的对象的状态。在默认的情况下，三种机制的对象都输出。下面就是命令行 ipcs 的某个输出结果。

```
$ ipcs

------ Shared Memory Segments --------
key          shmid        owner        perms        bytes        nattch       status
0x00000000 229376       lxy          600          393216       2            dest

------ Semaphore Arrays --------
key          semid        owner        perms        nsems

------ Message Queues --------
key          msqid        owner        perms        used-bytes   messages
```

从中可以看出，整个系统中只有一个共享内存段，它的关键字为 0x00000000，引用标识符为 229376，所有者为 lxy，权限为 600，即-rw—— 占用字节数为 393216。

6.4.2　消息队列

消息队列是一条由消息连接而成的链表，它保存在内核中，通过消息队列的引用标识符来访问。在本节中，我们把消息队列的引用标识符简单地称作队列标识符。

每个消息队列都有一个 msqid_ds 结构与之对应，在这个结构中，保存了消息队列的当前状态参数。这个结构的定义如下：

```
struct msqid_ds
```

```
{
 struct ipc_perm msg_perm;
 struct msg        *msg_first;
 struct msg        *msg_last;
 ulong             msg_ctypes;
 ulong             msg_qnum;
 ulong             msg_qbytes;
 pid_t             msg_lspid;
 pid_t             msg_lrpid;
 time_t                    msg_stime;
 time_t                    msg_rtime;
 time_t                    msg_ctime;
}
```

这个结构中，每个域的含义如表 6-7 所示。

表 6-7　msqid_s 结构每个域的含义

域	含　义
msg_perm	每个进程通信机制的对象都有个相对应的 ipc_perm 结构。这就是与这个队列相对应的 ipc_perm 的指针
msg_first 和 msg_last	队列中第一个消息和最末一个消息的指针，指向了对应的消息在内核中的位置，对用户进程来说一般没用
msg_cbytes、msg_qnum 和 msg_qbytes	无符号长整数，分别记录了当前队列中的字节总数、当前队列中消息的个数和队列中可存放的最大字节数
msg_lspid 和 msg_lrpid	分别记录最近一个执行 msgsnd 的进程的进程号和最近一个执行 msgrcv 的进程的进程号
msg_stime、msg_rtime 和 msg_ctime	分别记录最近一次执行 msgsnd 的时间、最近一次执行 msgrcv 的时间和最近一次消息队列改变的时间

系统中对于 System V 的三种机制都有一些限制，这些限制值可以在配置系统时改变，一旦系统配置完成，这些限制值就不可以改变了。与消息队列有关的限制如表 6-8 所示。

表 6-8 与消息队列有关的限制

值	说　明
MSGMAX	可以发送的消息的最大字节数
MSGMNB	一个队列中最大的字节总数
MSGMNI	系统中允许存在的最大消息队列个数
MSGTQL	系统中允许存在的最大消息个数

6.4.2.1 创建或打开消息队列的操作

通过系统函数调用，可以创建或打开消息队列，把一个消息发送到消息队列中，以及从消息队列中取得消息，下面详细地介绍这些操作。

首先看一下打开或创建消息队列的函数。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msq.h>
int msgget(key_t key, int flag);
```

这个函数可以创建一个新的消息队列，也可以用来打开一个已存在的消息队列，这取决于 key 和 flag 的值。

- 当 key 的取值为 IPC_PRIVATE 时，不管 flag 为何值，这个函数将创建一个新的消息队列。
- 当 key 的取值不为 IPC_PRIVATE 时，操作类型就取决于 flag 的值。如果 flag 中设置了 IPC_CREAT 位，而没有设置 IPC_EXCL 位，就既可能执行打开操作，也可能执行创建操作；当 key 的取值与内核中某个存在的消息队列的关键字相同时，执行打开这个消息队列的操作，返回引用标识符；反之，当 key 的取值不与存在的任何一个消息队列的关键字相同时，就会执行创建消息队列的操作，返回引用标识符。
- 当 key 的取值不是 IPC_PRIVATE 时，如果 flag 中同时设置了 IPC_CREAT 和 IPC_EXCL 两位，则只会执行创建消息队列操作：当 key 的取值不与存在的任何一个消息队列的关键字相同时，就会执行创建消息队列的操作，返回引用标识符；当 key 的取值与内核中某个存在的消息队列的访问键相同时，这个函数就会出错返回。

所以，打开存在的消息队列的方法只有一种：将 key 取为要打开的消息队列的关键字的值，而 flag 中绝对不能设置 IPC_EXCL 位，那么就会成功地打开这个消息队列。

另外，也可以通过 flag 参数设置消息队列的访问权限，通过后面的例子可以看到。

函数执行成功时会返回消息队列的引用标识符，否则返回-1。

当一个新的消息队列创建时，与之对应的 msqid_ds 结构会被初始化：

(1) ipc_perm 结构会被初始化，其中，mode 域的设置会按照 flag 的要求进行。

(2) msg_qunm、msg_lspid、msg_lrpid、msg_stime 和 msg_rtime 都会被置 0。

(3) msg_ctime 被置为当前时间。

(4) msg_qbytes 被置为系统限制值。

例 6-13 下列程序根据用户输入关键字，以权限-rw-rw---创建或打开一个消息队列。

```
1    /* ex13.c */
2    #include <stdio.h>
3    #include <sys/types.h>
4    #include <sys/ipc.h>
5    #include <sys/msg.h>
```

```
6
7    int main()
8    {
9      key_t key;
10     int msqid;
11
12     printf("Enter the desired key in hex =");
13     scanf("%x",&key);
14
15     printf("\nkey=0x%x", key);
16     if((msqid=msgget(key,IPC_CREAT|0660))==-1)
17     {
18          printf("The msgget failed.\n");
19          exit(1);
20     }
21     printf("The msgget succeeded:msqid=%d \n",msqid);
22     exit(0);
23   }
```

程序的第 12~13 行是让用户输入期望的关键字，随后调用 msgget 函数根据刚才的输入创建或打开消息队列(第 16~20 行)。程序的执行结果如下：

```
$ ./ex13
Enter the desired key in hex =0                /*0 为输入期望的关键字值 */

key=0x0The msgget succeeded:msqid=98307
$ipcs –q                                        /*查看系统中的消息队列*/

------ Message Queues --------
key          msqid      owner      perms      used-bytes    messages
0x00000000 98307        lxy        660        0             0
```

从 ipcs 的命令执行结果可以看出，系统中根据用户输入的关键字值成功创建了一个新的消息队列。

6.4.2.2 发送和接收消息

消息队列允许 2 种操作：发送消息和接收消息。进程通过向消息队列发送消息和从消息队列接收消息实现进程间的通信。

在介绍发送和接收消息的函数之前，先了解一下消息的组成。每个消息由两部分组成，消息的类型域和所传递的数据域。其中，类型域由一个正的长整数构成，而数据域根据不同的需要可以有不同的形式。例如传递的数据由字符组成，且一次传递，最长不会超过 512 个字符，那么消息可以定义为下面的类型：

```
struct mymsg
{
 long mtype;
 char mtext[512];
}
```

发送消息的函数说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

其中每个参数的含义为：

(1) msqid：消息队列的引用标识符。新发送的消息插入消息队列的末尾。

(2) ptr：指向一个长正整数的指针，这个正整数之后紧跟着消息中所传递的数据。举例说明，ptr 可以是指向我们自定义的结构 mymsg 的指针。

(3) nbytes：消息的长度，不包括那个长正整数在内，以字节记。

(4) flag：它可以取 0 或者 IPC_NOWAIT。一旦 IPC_NOWAIT 位被设置，在消息队列满的情况下(可能是消息个数太多，或者消息的总字节数太多)，msgsnd 将不等待而直接带错返回；否则，发送消息的函数将被阻塞，直到(a)消息队列腾出空间，或者(b)消息队列被删除。

函数成功返回 0，否则返回-1。

从消息队列中接收消息的系统函数说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

其中，参数的含义和 msgsnd 类似，分别为：

(1) msqid：消息队列的引用标识符。

(2) ptr：指向一个长正整数的指针，这个正整数之后紧跟着一块空间，用来存储消息中所传递的数据。如果函数调用成功，系统就会将消息复制到这块空间中，并将消息本身从队列中删除。

(3) nbytes：要接收的消息数据的长度，以字节记。消息中数据的长度超过了这个值就根据 flag 的值分两种情况来处理，要么截断数据，要么出错返回。

(4) type：这个参数用来指定要接收消息队列中的哪条消息，并不是按照先进先出的顺序。根据 type 的取值，可分为三种情况，如表 6-9 所示。

表 6-9　type 的取值说明

值	说　　明
0	返回消息队列中的第一个消息
>0	返回消息队列中的类型域等于这个值的第一个消息
<0	返回消息队列中的类型域小于等于 type 的绝对值的消息中，类型域最小的第一个消息

(1) flag：flag 中的两位与接收消息有关。

- 如果 IPC_NOWAIT 位被设置，在指定的 type 无效的情况下，msgrcv 将不等待而直接带错返回；否则 msgrcv 调用将被阻塞，直到(a)所希望的消息已放置在队列中，type 变为有效。(b)消息队列被删除。
- 如果 MSG_NOERROR 位被设置，当消息数据长度超过 nbytes 时，消息数据就被截断，函数正确返回；否则，函数错误返回，消息仍然存在于消息队列中。

msgrcv 调用成功，将返回接收的消息数据的字节数，否则返回-1。同时将更新与消息队列 msqid 相连数据结构的成员：msg_qnum 减少 1，msg_lrpid 等于调用进程的进程 ID，msg_rtime 等于当前时间。

现在可以编写用消息队列进行通信的程序了。我们编写两个程序，程序 14 从消息队列接收消息，程序 15 则发送消息。每一个消息是用户输入的任意字符串，字符串 end 表示输入结束。

```
1   /* ex14.c */
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <string.h>
5   #include <unistd.h>
6   #include <sys/types.h>
7   #include <sys/ipc.h>
8   #include <sys/msg.h>
9
10  struct my_msg
11  {
12    long int my_msg_type;
13    char text[BUFSIZ];
14  } msgbuf;
15
16  int main()
17  {
18    int running =1;
19    int msgid;
20    long int msg_to_receive=0;
```

```
21  msgid=msgget((key_t)1234,0666 |IPC_CREAT);
22  if(msgid==-1)
23  {
24          printf("msgget failed!\n");
25          exit(1);
26  }
27  while(running)
28  {
29          if(msgrcv(msgid,(void *)&msgbuf, BUFSIZ,msg_to_receive, 0)==-1)
30          {
31                  printf("msgrcv failed!\n");
32                  exit(1);
33          }
34          printf("You wrote : %s", msgbuf.text);
35          if(strncmp(msgbuf.text,"end",3)==0)
36                  running=0;
37  }
38  if(msgctl(msgid, IPC_RMID, 0)==-1)
39  {
40          printf("msgct(IPC_RMID)    failed!\n");
41          exit(1);
42  }
43  return 0;
44 }
```

说明：程序 14 完成消息接收的功能。程序首先定义了消息结构(第 10~14 行)，在 main 函数里先打开发送程序(程序 15)创建的消息队列，消息队列的关键字设为 1234(第 21 行)，然后从队列中按正常顺序、阻塞方式反复接收消息，然后显示收到的消息，直到收到字符串 end(第 27~37 行)，然后用 msgctl 函数删除队列(第 38~42 行)，程序返回。

下面是程序 15 的代码：

```
1   /* ex15.c */
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <string.h>
5   #include <unistd.h>
6   #include <sys/types.h>
7   #include <sys/ipc.h>
8   #include <sys/msg.h>
9
10  struct my_msg
11  {
12    long int my_msg_type;
```

```
13   char text[BUFSIZ];
14  } msgbuf;
15
16  int main()
17  {
18   int running =1;
19   int msgid;
20   msgid=msgget((key_t)1234,0666 |IPC_CREAT);
21   if(msgid==-1)
22   {
23           printf("msgget failed!\n");
24           exit(1);
25   }
26   while(running)
27   {
28           printf("Enter some text: ");
29           fgets(msgbuf.text,BUFSIZ,stdin);
30           msgbuf.my_msg_type=1;
31           if(msgsnd(msgid,(void *)&msgbuf, BUFSIZ, 0)==-1)
32           {
33                   printf("msgsnd failed!\n");
34                   exit(1);
35           }
36           if(strncmp(msgbuf.text,"end",3)==0)
37                   running=0;
38   }
39   return 0;
40  }
```

说明：程序 15 的结构与程序 14 基本相同，它的功能是反复接收用户的输入，然后将输入的数据发送给消息队列，直至接收到 end 为止(第 26~38 行)。

先执行发送消息程序，即程序 15。下面是这两个程序的运行结果：

```
$ ./ex15
Enter some text: hello
Enter some text: world
Enter some text: end
$ ./ex14
You wrote : hello
You wrote : world
You wrote : end
```

从这个例子可以看出，消息队列不同于管道，通信的两个进程可以是完全无关的进程，

它们之间不需要约定同步的方法。只要消息队列存在并且有存放消息的空间，发送进程就可以向队列中存放消息，并且可以在接收进程开始之前终止其执行。但使用管道通信的进程，不论是匿名管道还是有名管道，通信的两个进程都必须是正在运行的进程。这一点正是消息队列的优点。

6.4.2.3 控制消息队列

msgctl 函数用于对消息队列执行如下控制操作：

(1) 查看消息队列相连的数据结构。

(2) 改变消息队列的许可权限。

(3) 改变消息队列的拥有者。

(4) 改变消息队列的字节大小。

(5) 删除一个消息队列。

它的原型如下：

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

msgctl 函数对 msqid 指定的消息队列执行参数 cmd 要求的控制操作。参数 msqid 是一个正整数，它必须是由 msgget 返回的消息队列的 id。参数 cmd 指定要求的操作，所允许的操作如表 6-10 所示。

表 6-10　cmd 指定的操作

值	操　作
IPC_STAT	复制 msqid 指定的消息队列的内核控制数据结构 msqid_ds 至 buf 所指的用户区域中
IPC_SET	设置 msqid 指定的消息队列的有效用户与组 ID、操作权限、以及消息队列的字节数，即设置 msqid 相连的数据结构个成员 msg_perm.uid、msg_perm.gid、msg_perm.mode 和 msg_qbytes 的值为 buf 所指结构中给出的值
IPC_RMID	删除 msqid 以及它所指定的消息队列和相连的数据结构

执行 IPC_STAT 命令的进程必须具有消息队列的读权限。执行 IPC_SET 和 IPC_RMID 命令的进程只能是消息队列的创建者、拥有者或特权进程。换言之，执行这两个命令的非特权进程必须是有效用户 ID 等于相连数据结构的成员 msg_perm.cuid 或 msg_perm.uid 的进程。此外，只有特权进程才可以增大消息队列的字节数。

参数 buf 指向类型为 msqid_ds 的结构，该结构由用户分配存储空间，它用于存放 IPC_STAT 命令的返回结果，或 IPC_SET 命令要设置的值。

如果调用成功，msgctl 返回 0，否则返回-1。

消息队列总是保持在系统中直到系统重启，除非我们明确地删除它。当不再需要一个消息队列时，应当用 IPC_RMID 命令删除它。消息队列一旦删除，那些当前被阻塞并企图

读写它的进程将欲唤醒并返回 EIDRM 错误。

在上一小节程序 14 中，演示了用 msgctl 函数删除消息队列的方法，此处不再举例。

6.4.3　信号量

信号量并不类似我们前面介绍的几种通信机制，它实际上是个整数计数器，主要用来控制多个进程对共享资源的访问。共享资源分为两类，一类为互斥共享，即某一时刻只能允许一个进程对资源进行操作；另一类为同步共享，同一时刻可以有若干进程对其进行某种操作。信号量就是用来帮助实现多进程对资源的共享的机制。它比较难以理解，我们先来看几个形象的例子。

信号量的名字来源于十字路口的信号灯。当红灯亮时，南北车辆通过路口，东西车辆等待：而绿灯亮时，东西车辆通过路口，南北车辆等待。由此，假设有某个共享资源，某一时刻只能允许一个进程对其进行操作，就像路口只能允许一个方向的车辆通过一样，是一种互斥资源。此时，信号量就像红绿灯一样，当某个进程对资源操作时，把它设置为一种形式，锁定资源，不允许其他进程使用；当这个进程完成操作后，释放资源，把它设置为另外一种形式，允许其他进程使用。这就是比较典型的信号量的使用形式，用于协调多个进程使用同一互斥资源。

信号量还有一种使用形式，用于处理多个共享资源。比如有六台打印机，若干人要使用。打印机总管手里有空闲打印机个数的记录：当有人要使用打印机时，总管查看空闲打印机数目记录，如果大于零，就可以将打印机资源分配出去，空闲打印机数减一，否则请使用者等待；使用者用完打印机后归还时，空闲打印机数加一，若有使用者等待，就将打印机分配出去。信号量在对多个共享资源的控制中，就起到记录空闲资源数目的作用。

现在读者应该对信号量有了一个形象的印象，下面开始介绍 Linux 系统中对信号量的一些规定。

当一个进程要访问某个共享资源时，它按下列步骤进行：

(1) 检测控制这个资源的信号量的值。

(2) 如果信号量的值是正数，就可以使用这个资源。进程将信号量的值减一，表示它正在使用资源的某个小单元。

(3) 如果信号量的值为零，那么这个进程进入睡眠状态，直到信号量的值重新大于零时被唤醒，转入第一步操作。

为了正确地实现信号量这一机制，检测和增减信号量的值应该是原子操作，所以信号量一般是在内核中实现的。

有一种信号量的普通形式，叫做二元信号量。它只控制一个资源，信号量的初始值设为 1。这就是前面介绍的十字路口信号灯的情况。普遍来讲，信号量的初值可以是任意的正整数，这个初值就是共享资源可以提供的可供共享的单元的个数。

与消息队列相同，每个信号量都与一个结构相联系，这个结构的定义在头文件

<sys/sem.h>中。具体描述如下：

```
struct semid_ds
{
        struct ipc_perm   sem_perm;
        struct sem        *sem_base;
        ushort                 sem_nsems;
        time_t                 sem_otime;
        time_t                 sem_ctime;
}
```

结构中的域的含义如表 6-11 所示。

表 6-11　结构 semid_ds 每个域的含义

域	含　　义
sem_perm	与消息队列相同，这个指针指向与这个信号量集相对应的 ipc_perm 结构的指针
sem_base	指向这个集合中第一个信号量的位置的指针，这个域对于用户进程是没有用处的，它实际指向一个 sem 结构的数组，数组中有 sem_nsems 个元素，每个元素对应信号量集中的一个信号量
sem_nsems	集合中信号量的个数
sem_otime	最近一次调用 semop 函数的时间
sem_ctime	最近一次改变的时间

下面介绍 semid_ds 中涉及到的 sem 结构，这个结构中记录了单一信号量的一些信息，具体描述如下：

```
struct sem
{
        ushort      semval;
        pid_t       sempid;
        ushort      semncnt;
        ushort      semzcnt;
}
```

其中每个域的含义如表 6-12 所示。

表 6-12　结构 sem 每个域的含义

域	含　　义
semval	信号量的值
sempid	最近一次执行操作的进程的进程号
semncnt	等待信号值增长，即等待可利用资源出现的进程数
semzcnt	等待信号值减少到零，即等待全部资源可被独占的进程数

和消息队列相同，信号量也有一些与之相关的系统限制，它们如表 6-13 所示。

表 6-13　与信号量相关的系统限制

值	说　明
SEMVMX	最大的信号值
SEMMNI	系统中允许的最大的信号量集的个数
SEMMNS	系统中允许的最大的信号量的个数
SEMMSL	每个信号量集中最大的信号量的个数

6.4.3.1　创建或打开信号量集

创建或打开信号量集的系统函数原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

其中，参数 key、flag 的含义和用途与创建或打开消息队列的函数 msgget 中是相同的。int nsems 没有在 msgget 的参数中出现，它指出信号量集中应该创建的信号量的数量。如果是打开一个已存在的信号量集，这个参数就被忽略。

当一个新的信号量集被创建的同时，与之相关联的 semid_ds 结构被初始化：

(1) ipc_perm 结构会被初始化，其中，mode 域的设置会按照 flag 的要求进行。

(2) sem_otime 被置为零。

(3) sem_ctime 被置为当前值。

(4) sem_nsems 被置为参数 nsems 的值。

semget 调用成功返回信号量集合标识符，否则返回-1。

例 6-14　用 semget 函数编制一个创建或打开信号量集的函数。

```
1   /* ex16.c */
2   int open_semaphore_set(key_t keyval, int numsems)
3   {
4     int sid;
5     if(!numsems)
6           return -1;
7     if((sid=semget(keyval, numsems,IPC_CREAT | 0660))==-1)
8           return -1;
9     else
10          return sid;
11  }
```

说明：这个函数以-rw-rw---模式创建或打开一个信号量集(第 7 行)。当 keyval 不与某

个存在的信号量集相联系时，这个函数创建一个新的信号量集，否则它会打开这个已存在的信号量集。函数出错返回-1(第 8 行)，否则返回信号量集标识符(第 10 行)。

6.4.3.2　信号量操作

函数 semop 用于操作信号量集合，它的说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf* sops, size_t nops);
```

其中，semid 为信号集的标识符；sops 为 sembuf 结构的数组，nops 为数组中元素的个数。而 sembuf 结构中记录了对信号集的一个操作，它的具体说明如下：

```
struct sembuf
{
        ushort      sem_num;
        short       sem_op;
        short       sem_flg;
}
```

其中，每个域的含义如表 6-14 所示。

表 6-14　结构 sembuf 每个域的含义

域	含　义
sem_num	要处理的信号量在信号量集中的序号
sem_op	它可以取正值、负值或者零，表示了要执行的操作
sem_flg	操作标记，相关的有 IPC_NOWAIT 和 SEM_UNDO 这两个标志

下面详细说明不同的 sem_op 代表的含义。

最简单的是 sem_op 为正数的情况。这对应于进程使用完资源，交回资源的情景。信号量的值将加上 sem_op 的值。如果 sem_flag 中的 SEM_UNDO 位被置为 1，那么信号量的调整值就会减去 sem_op 的值。

如果 sem_op 为负数，表示进程希望使用资源。这时就要视可用资源的多少而不同了：

(1) 如果信号的值不小于 sem_op 的绝对值，表示可用资源足够分给这个进程的，那么，就要从信号的值中减去 sem_op 的绝对值，表示分配给进程那么多资源。如果 sem_flag 中的 SEM_UNDO 位被置为 1，那么信号量的调整值就会加上 sem_op 的绝对值。

(2) 如果信号的值小于 sem_op 的绝对值，表示资源不够了，那么根据 sem_flag 的值有不同的操作：

- 如果 sem_flag 中的 IPC_NOWAIT 位被置为 1，这个函数就会立即带错返回。

- 如果 sem_flag 中的 IPC_NOWAIT 没有被置位，与这个信号量相关的 sem 结构中的 semncnt 域的值加一，这个进程进入睡眠状态，直到其他进程返回了资源，信号量的值不小于 sem_op 的绝对值。那么进程就被唤醒，semncnt 的值减一，转入第一种情况的处理：或者这个信号量被删除，在这种情况下，这个函数带错返回。

如果 sem_op 为零，这表示了进程要一直等待，直到信号量的值变为零。

(3) 如果信号量的值恰好为零，函数立即返回。

(4) 如果信号量的值不为零，与这个信号量相关的 sem 结构中的 semzcnt 域的值加一，这个进程进入睡眠状态，直到信号量的值变成了零，那么进程就被唤醒，semzcnt 的值减一；或者这个信号量被删除，在这种情况下，这个函数带错返回。

这个函数中的所有操作是一个原语，要么完成整个数组中记录的操作，要么一个操作也不执行。

函数调用成功返回 0，否则返回-1。

信号量最常见的用法是控制程序中的一个关键区。这个关键区可能需要访问由多个进程共享的数据段，也可能需要访问其他共享资源，比如说打印机。在任一时刻，应当只有一个进程执行关键区的代码，从而保证只有一个进程访问对应的共享资源。因此，当一个进程要进入关键区时，它应当执行 P 操作(使信号量值减少)；当它离开关键区时．要执行 V 操作(使信号量值增加)。下面看一个例子。

例 6-15 信号量用法的简单程序。这个程序可以同时多次运行，它所做的工作只是输出命令行参数，每个参数输出一行。不过它很懒，每输出一个字符，它便要睡眠一会儿，睡眠时间由随机数提供。用这种方式工作，如果这个程序同时有两个进程执行它，那么就有可能导致同一行的输出来自不同的进程。为了保证每一行的输出内容来自同一个进程，完成一个参数输出工作的代码必须作为关键区。代码如下：

```
1  /* ex17.c */
2  #include <sys/types.h>
3  #include <sys/ipc.h>
4  #include <sys/sem.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <stdio.h>
8
9  int semaphore_P(int);
10 int semaphore_V(int sem_id);
11
12 int main(int argc, char* argv[])
13 {
14   int sem_id;
15   int i, creat=0;
16   int pause_time;
```

```
17  char *cp;
18
19  if(argc<=1)
20  {
21          printf("usage: %s parameter1 parameter2 ...\n",argv[0]);
22          exit(1);
23  }
24  srand((unsigned int)getpid());
25  if((sem_id=semget((key_t)1234,1,IPC_CREAT | 0660))==-1)
26  {
27          printf("semget failed!\n");
28          exit(2);
29  }
30  if(strcmp(argv[1],"1"))
31  {
32          semctl(sem_id, 0, SETVAL, 1);
33          creat=1;
34          sleep(2);
35  }
36  for(i=0; i<argc;i++)
37  {
38          cp=argv[i];
39          if(!semaphore_P(sem_id))
40                  exit(3);
41          printf("Process %d:", getpid());
42          fflush(stdout);
43          while(*cp)
44          {
45                  printf("%c", *cp);
46                  fflush(stdout);
47                  pause_time=rand() %3 ;
48                  sleep(pause_time);
49                  cp++;
50          }
51          printf("\n");
52          if(!semaphore_V(sem_id))
53                  exit(4);
54          pause_time=rand() %2 ;
55          sleep(pause_time);
56  }
57  printf("\n %d -finished \n",getpid());
58  if(creat==1)
59  {
```

```
60              sleep(10);
61              semctl(sem_id,0,IPC_RMID,0);
62      }
63      return 0;
64  }
65
66
67  int semaphore_P(int sem_id)
68  {
69      struct sembuf sb;
70      sb.sem_num=0;
71      sb.sem_op=-1;
72      sb.sem_flg=SEM_UNDO;
73
74      if(semop(sem_id, &sb, 1)==-1)
75      {
76              printf("semaphore_P failed.\n");
77              return 0;
78      }
79      return 1;
80  }
81
82  int semaphore_V(int sem_id)
83  {
84      struct sembuf sb;
85      sb.sem_num=0;
86      sb.sem_op=1;
87      sb.sem_flg=SEM_UNDO;
88      if(semop(sem_id, &sb, 1)==-1)
89      {
90              printf("semaphore_V failed. \n");
91              return 0;
92      }
93      return 1;
94  }
```

说明：程序首先创建一个仅包含一个信号量的集合(第 25~29 行)，当命令行中输入的第 1 个参数为“1”时，用 semctl 函数(这个函数下一小节介绍)设置第一个信号量的初值为1(第 30 ~35 行)。程序第 41 行是关键区的开始，第 51 行是关键区的结束。关键区所做的工作只是输出命令行参数，每个参数输出一行。函数 semaphore_P(第 67~80 行)和semaphore_V(第 82~94 行)分别对该信号量执行 P/V 操作，在进入关键区时调用semaphore_P(第 39 行)，离开关键区时调用 semaphore_V(第 52 行)，由此保证每次只有一

个进程执行关键区代码。

我们约定命令行的第一个参数为 1(第 30 行)表示这是负责给信号量赋初值和删除信号量集合的进程。同时，为了演示的需要，程序在删除信号量集合之前睡眠 10 秒钟(第 60 行)，以便另一个进程在此之前退出。

我们先在后台运行这个程序，同时指定它的第一个参数为 1，该参数为 1 表示要创建信号量。然后再以其他参数运行该程序，结果如下所示：

```
$ ./ex17 1 test semaphore &          /*以后台方式执行程序 */
[1] 7443
7$ ./ex17 Hello world!
Process 7489:./ex17
Process 7490:./ex17
Process 7489:1
Process 7490:Hello
Process 7489:test
Process 7490:world!
Process 7489:se
 7490 -finished
maphore

 7489 -finished
[1]+  Done                      ./ex17 1 test semaphore
```

可以看到，除最后一个输出参数之外，每一个参数都如希望的那样输出在同一行中。但两个进程的参数却是随机混杂在一起的，因为我们只用信号量控制了每一个参数的输出，没有控制所有参数的输出。最后一个参数的输出被隔断是因为语句

```
printf("\n %d -finished \n",getpid());
```

没有处在关键区内，它不受信号量的控制，它的输出内容使得单词 semaphore 出现在不连续的位置。

同消息队列的情形一样，在实际应用中，应注意删除所使用的信号量集合。因为如果不明确删除，它们将保持在系统内直至系统重启。

6.4.3.3 信号量控制

调用 semget 只能创建一个信号量集合，但这个信号量集合中每一个信号量并没有初值。例如，如果我们希望用一个信号量管理含有 20 个缓冲区的一个缓冲池，则这个信号量的初值应当置为 20。为了给信号量集合中每一个信号量置一个初值，需要调用 semctl 函数。

semctl 函数称为信号量控制函数，除了设置信号量初值之外，它还可以获取与信号量集合相连数据结构 semid_ds，改变信号量集合拥有者以及访问权限，删除指定的信号量集合，查看与信号量集合有关的其他信息，如最后一个操作它的进程和在该信号量集合上等

待的进程数等。

semctl 的函数原型为：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, [union semun arg]);
```

第一个参数 semid 必须是一个合法的信号量集合标识；第 2 个参数 semnum 选择集合中一个特定的信号量，它指明这个信号量的编号；第 3 个参数给出操作命令，表 6-15 列出了 semctl 可以执行的所有命令；第 4 个参数 arg 是可选的，它是否需要取决于所执行的命令。当执行的是表上半部分的命令时，必须提供该参数。

参数 arg 是一个类型为 semun 的联合。当需要时，应用程序必须这样定义它：

```
union semun
{
        int val;
        struct semid_ds *buf;
        unsigned short *array;
} arg;
```

其中，val 用于 SETVAL 命令，指明要设置的信号量值；buf 用于 IPC_STAT/IPC_SET 命令，表示存放信号量集合数据结构的缓冲区；array 用于 GETALL/SETALL 命令，存放所获得的或要设置的信号量集合中所有信号量的值。

表 6-15　semctl 命令

命　　令	说　　明
SETVAL	设置单个信号量的值
GETALL	返回信号量集合中所有信号量的值
SETALL	设置信号量集合中所有信号量的值
IPC_STAT	放置与信号量集合相连的 semid_ds 结构当前值于 arg.buf 指定的缓冲区
IPC_SET	用 arg.buf 指定结构值替代与信号量集合相连的 semid_ds 结构值
GETVAL	返问单个信号量的值
GETPID	返回最后一个操作该信号量集合的进程 ID
GETNCNT	返回 semncnt 之值
GETZCNT	返回 semzcnt 之值
IPC_RMID	删除指定的信号量集合

执行 IPC_SET、IPC_RMID 命令的进程只能是信号量集合的创建进程、拥有进程或特权进程，执行其他命令的进程必须有信号量集合的读或更新权限。

semctl 的返回值在调用成功时，对于 GETVAL、GETPID、GETNCNT 和 GETZCNT 是命令的要求值，对于其他命令，返回值为 0；调用失败，返回值为-1。

在上一小节中已经给出了 semctl 函数使用方法的例子，此处不再另外举例。

6.4.4 共享内存

共享内存是允许多个进程共享一块内存，由此来达到交换信息的进程通信机制。共享内存机制是最快的一种进程通信机制，因为没有中间介质，如消息队列、管道等等的延迟，数据直接由内存映射到进程空间。通常，共享内存段由一个进程创建，接下来的读写操作就由许多进程参加，这样就能传递信息了。

共享内存机制唯一的不足在于，需要一定的同步机制控制多个进程对同一块内存的读写。当一个进程在写数据时，不允许其他的进程写数据或读数据，这可以通过信号量控制实现。

同前面的两种通信机制一样，每个共享内存段都对应一个 shmid_ds 结构。这个结构的定义如下：

```
struct shmid_ds
{
 struct ipc_perm  shm_perm;
 int              shm_segsz;
 ushort           shm_lkcnt;
 pid_t            shm_cpid;
 pid_t            shm_lpid;
 ulong            shm_nattach;
 time_t           shm_atime;
 time_t           shm_dtime;
 time_t           shm_ctime;
};
```

其中每个域的含义如表 6-16 所示。

表 6-16 shmid_ds 结构每个域的含义

域	含 义
sem_perm	与信号量相同，这个指针指向与这个共享内存相对应的 ipc_erm 结构的指针
shm_segsz	共享内存段的大小，以字节记
shm_lkcnt	共享内存段被锁定的时间数
shm_lpid	最近一次调用 shmop 函数的进程的进程号
shm_cpid	创建这个共享内存段的进程的进程号
shm_nattach	当前把这个内存段附加到地址空间的进程数

(续表)

域	含　义
shm_atime	最近一次附加操作的时间
shm_dtime	最近一次分离操作的时间
shm_ctime	最近一次改变的时间

像前面两种 System V 机制一样，有一些系统限制是与共享内存机制相关的，如表 6-17 所示。

表 6-17　与共享内存相关的系统限制

值	说　明
SHMMAX	共享内存段的最大字节数
SHMMIN	共享内存段的最小字节数
SHMMNI	系统中允许存在的共享内存的最大个数
SHMSEG	一个进程中允许存在的共享内存的最大个数

下面详细介绍有关共享内存的函数调用。

6.4.4.1　共享内存的创建与打开

要使用共享内存，首先要创建一个共享内存区域。创建共享内存的函数如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int flag);
```

函数 shmget 除可用于创建一个新的共享内存外，也可用于打开一个已存在的共享内存。其中，参数 key 表示所创建或打开的共字内存的关键字。参数 size 表示共享内存区域的大小，只在创建一个新的共享内存时生效。参数 flag 表示调用函数的操作类型，也可用于设置共享内存的访问权限，两者通过逻辑或表示。调用函数 shmget 的作用由参数 key 和 flag 决定，相应约定如下：

(1) 当 key 为 IPC_PRIVATE 时，创建一个新的共享内存。此时参数 flag 的取值对函数的操作不起任何作用。

(2) 当 key 不为 IPC_PRIVATE，且 flag 设置了 IPC_CREAT 位，而没有设置 IPC_EXCL 位，则执行操作由 key 取值决定。如果 key 为内核中某个已存在的共享内存的键，则执行打开这个键的操作；反之，则执行创建共享内存的操作。

(3) 当 key 不为 IPC_PRIVATE，且 flag 中同时设置了 IPC_CREAT 位和 IPC_EXCL 位，则只执行创建共享内存操作。参数 key 的取值应与内核中已存在的任何共享内存的关键字

都不相同，否则函数调用失败。

此函数调用成功时，返回值为共享内存的引用标识符；调用失败时，返回值为-1。

当调用 shmget 函数创建一个共享内存时，此共享内存的 shmid_ds 结构被初始化。ipc_perm 中的各个域被设置为相应值，shm_lpid、shm_nattach、shm_atime 和 shm_dtime 被设置为 0，shm_ctime 设置为当前时间。

例 6-16 用 shmget 函数编制一个创建或打开一块新共享内存的函数。

```
1   /* ex18.c */
2   #include <sys/types.h>
3   #include <sys/ipc.h>
4   #include <sys/shm.h>
5
6   int openshm(int size)
7   {
8     int shmid;
9     if(shmid=shmget(IPC_PRIVATE,size,0)==-1)
10    {
11          printf(" Get shared memory failed!\n");
12          return -1;
13    }
14    return shmid;
15  }
```

说明：这个函数创建或打开一块新的共享内存(第 9 行)。函数出错返回-1(第 12 行)，否则返回信号量集标识符(第 14 行)。

6.4.4.2　共享内存的操作

当一个共享内存创建或打开后，某个进程如果要使用该共享内存则必须将此内存区域附加到它的地址空间。附加操作的相关调用如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat(int shmid, void *addr, int flag);
```

其中，参数 shmid 表示要附加的共享内存段的引用标识符。参数 flag 用于表示 shmat 函数的操作方式。如果 flag 设置了 SHM_RDONLY 位，该内存区域被设置为只读，否则设置为可读写。参数 addr 和 flag 共同决定共享内存区域要附加到的地址值，相应约定如下：

(1) 如果 addr 为 0，系统将自动查找进程地址空间、将共享内存区域附加到第一块有效内存区域上，此时 flag 无效。

(2) 如果 addr 不为 0，而 flag 未设置 SHM_RND 位，则共享内存区域附加到由 addr 指定的地址处。

(3) 如果 addr 不为 0，而 flag 设置了 SHM_RND 位。则共享内存区域附加到由 addr-(addr %SHMLBA)指定的地址处。

调用成功时，返回值为共享内存区域的指针；调用失败时，返回值为-1。

当一个进程对共享内存区域的访问完成后，可以调用 shmdt 函数使共享内存区域与该进程的地址空间分离。shmdt 函数的说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(void *addr);
```

此函数仅用于将共享区域与进程的地址空间分离，并不删除共享内存本身。参数 addr 为要分离的共享内存区域的指针，是调用 shmat 函数时的返回值。调用成功时，返回值为 0；调用失败时，返回值为-1。

6.4.4.3 共享内存的控制

对共享内存区域的具体控制操作是通过函数 shmctl 来实现的。shmctl 函数的说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, shmid_ds *buf);
```

其中，参数 shmid 为共享内存的引用标识符。参数 cmd 表示调用该函数希望执行的操作。参数 buf 是指向 shmid_ds 结构的指针。参数 cmd 的取值和对应操作如表 6-18 所示。

表 6-18 cmd 的取值和对应操作

值	操 作
SHM_LOCK	将共享内存区域上锁。只能由超级用户执行
IPC_RMID	用于删除共享内存。执行该命令时，共享内存的引用标识符立刻被删除，则该共享内存不能再被其他进程所附加。但共享内存的真正删除要到所有附加了该共享内存的进程结束或断开与该共享内存的连接时才执行
IPC_SET	按参数 buf 指向的结构中的值设置该共享内存对应的 shmid_ds 结构。只有有效用户 ID 和共享内存的所有者 ID 或创建者 ID 相同的用户进程，以及超级用户进程可以执行这一操作
IPC_STAT	用于取得该共享内存的 shmid_ds 结构，保存于 buf 指向的缓冲区
SHM_UNLOCK	将上锁的共享内存区域释放。只能由超级用户执行

例 6-17 用共享内存实现客户-服务器的通信模式。

首先实现服务器程序。

```
1    /* ex19 server.c*/
2    #include <sys/types.h>
3    #include <sys/ipc.h>
4    #include <sys/shm.h>
5    #include <stdio.h>
6    #include <string.h>
7
8    int main()
9    {
10     int     shmid;
11     char   c;
12     char   *shmptr, *s;
13     if((shmid=shmget(1234,256,IPC_CREAT | 0666))<0)
14     {
15             printf("shmget failed.\n");
16             exit(1);
17     }
18     if((shmptr=shmat(shmid,0,0))==-1)
19     {
20             shmctl(shmid, IPC_RMID, shmptr);
21             printf("shmat failed.\n");
22             exit(2);
23     }
24     s=shmptr;
25     for(c='a';c<='z';c++)
26             *s++=c;
27     *s=NULL;
28     while(*shmptr!='*')
29             sleep(1);
30     shmctl(shmid, IPC_RMID, shmptr);
31     return 0;
32   }
```

说明：服务程序首先创建一个名为“1234” 的共享内存并连接至自己的地址空间(第13~23 行)。完成这两步操作后，便可以往共享内存写数据了。写共享内存通过指针操作即可。写入的数据很简单，仅仅是“a” 至 z” 26 个字母(第 25~26 行)。当数据写完后，进程等待直到得知客户进程已读完数据。客户进程简单地置共享内存第一字节为“*”来通知服务进程数据已读完(第 28~29 行)。最后，这个服务程序删除共享内存，退出执行(第 30~31 行)。

下面是客户程序：

```
1    /*ex19 client.c*/
2    #include <sys/types.h>
3    #include <sys/ipc.h>
```

```
4   #include <sys/shm.h>
5   #include <stdio.h>
6   #include <string.h>
7
8   int main()
9   {
10
11    int     shmid;
12    char   c;
13    char   *shmptr, *s;
14    if((shmid=shmget(1234,256, 0666))<0)
15    {
16            printf("shmget failed.\n");
17            exit(1);
18    }
19    if((shmptr=shmat(shmid,0,0))==-1)
20    {
21            shmctl(shmid,IPC_RMID,shmptr);
22            printf("shmat failed.\n");
23            exit(2);
24    }
25    for(s=shmptr;*s!=NULL;s++)
26            putchar(*s);
27    printf("\n");
28    *shmptr='*';
29    return 0;
30  }
```

说明：客户程序读出服务进程写至共享内存的内容，并将它们打印出来(第 25~26 行)。它与服务进程有一点不同——打开名为“1234”的共享内存时，它没有指定 IPC_CREAT。这样，当服务进程没有运行时，由于该共享存储段不存在，客户此时将错误返回(第 14 行)。

下面是程序的运行结果，先在一个终端窗口中运行服务程序：

```
$ ./ex19server
```

运行后，光标就停在那里等待客户程序的执行，打开另一个终端窗口，运行客户程序：

```
$ ./ex19client
abcdefghijklmnopqrstuvwxyz
```

输出上述信息后，客户程序退出，同时第一个终端窗口中的服务程序也退出。

上面给出的例子在实际应用中是不可靠的，因为，客户进程有可能先于服务进程写完数据之前读共享存储段中的数据。实际应用中习惯的做法是使用信号量来协调客户与服务

进程之间对共享存储段的访问。我们将在下一节给出一个综合应用实例。

下面再看一个内存共享的例子。

例 6-18 当系统调用 shmat 中的 addr 取值为 0 时，共享内存附加到的地址是完全由系统决定的。编写一个程序，测试一下共享内存附加到的地址。

```
1  /*ex20.c */
2  #include <sys/types.h>
3  #include <sys/ipc.h>
4  #include <sys/shm.h>
5  #include <stdio.h>
6
7  char array[4000];
8
9  int main()
10 {
11   int     shmid;
12   char   *ptr, *shmptr;
13   printf("array[] form %x to %x \n",&array[0],&array[3999]);
14   printf("stack around %x \n", &shmid);
15   if((ptr=malloc(10000))==NULL)
16   {
17           printf("malloc failed.\n");
18           exit(1);
19   }
20   if((shmid=shmget(IPC_PRIVATE,10000,SHM_R|SHM_W))<0)
21   {
22           printf("shmget failed.\n");
23           exit(2);
24   }
25   if((shmptr=shmat(shmid,0,0))==-1)
26   {
27           printf("shmat failed.\n");
28           exit(3);
29   }
30   printf("shared memory attached from %x to %x \n",shmptr,shmptr-10000);
31   if(shmctl(shmid,IPC_RMID,0)<0)
32   {
33           printf("shmctl failed.\n");
34           exit(4);
35   }
36   return 0;
37 }
```

说明：由于 shmat 函数调用将共享内存段的地址作为返回值，只要将这个地址输出，就可以知道共享内存段附加的位置了。这段程序在作者的机器上的运行结果如下：

```
$ ./ex20
array[] form 80498c0 to 804a85f
stack around bfc36a48
shared memory attached from b7fdf000 to b7fdc8f0
```

从执行结果可以看出，共享内存段附加到了进程栈的下面。

6.4.5　综合应用实例

这一节给出一个综合实例设计，重点说明信号量和共享内存的使用。

例 6-19　下面是使用共享存储交换数据的另一个例子。其中一个程序是数据处理程序，另一个程序是数据生成程序。这两个程序使用两个信号量 consumer 和 producer 来同步它们之间对共享内存的访问。consumer 信号量指出共享存储段中的数据是否已被处理完毕，只有当处理完毕后才允许数据生成程序在其中继续生成数据；producer 指出数据是否已生成在共享内存。一个共享内存用来交换数据，所交换数据的格式由 exchange 结构描述。其中，成员 buf 用于存放数据，seq 存放客户进程写入的顺序号。两个程序都使用一个指向该共享存储段的指针对共享内存直接进行读写。下面是这两个程序都要用到的定义和说明，它们集中在名为 myshm.h 的头文件中。

```
1  /*ex21 myshm.h"*/
2  #include <sys/types.h>
3  #include <sys/ipc.h>
4  #include <sys/sem.h>
5  #include <sys/shm.h>
6  #include <stdio.h>
7  #include <string.h>
8
9  #define SHMSZ    256
10 union semun
11 {
12   int val;
13   struct semid_ds    * buf;
14   unsigned short     *array;
15 };
16
17 void init_a_semaphore(int sid, int semnum, int initval)
18 {
19   union semun semopts;
20   semopts.val=initval;
```

```
21    semctl(sid, semnum,SETVAL,semopts);
22  }
23
24  int semaphore_P(int sem_id)
25  {
26    struct sembuf sb;
27    sb.sem_num=0;
28    sb.sem_op=-1;
29    sb.sem_flg=SEM_UNDO;
30
31    if(semop(sem_id, &sb, 1)==-1)
32    {
33            printf("semaphore_P failed.\n");
34            return 0;
35    }
36    return 1;
37  }
38
39  int semaphore_V(int sem_id)
40  {
41    struct sembuf sb;
42    sb.sem_num=0;
43    sb.sem_op=1;
44    sb.sem_flg=SEM_UNDO;
45    if(semop(sem_id, &sb, 1)==-1)
46    {
47            printf("semaphore_V failed. \n");
48            return 0;
49    }
50    return 1;
51  }
```

myshm.h 中的几个函数前面已经介绍过，这里不再赘述。

下面是数据处理程序：

```
1   /*ex21 client.c*/
2   #include "myshm.h"
3
4   int main()
5   {
6     char *shm;
7     int shmid;
8     int producer, consumer, i;
```

```
9     /*创建和初始化信号量 consumer */
10    if((consumer =semget(ftok("consumer",0),1,IPC_CREAT|0660))==-1)
11    {
12          printf("semget failed.\n");
13          exit(1);
14    }
15    init_a_semaphore(consumer,0,1);
16    /*创建和初始化信号量 producer */
17    if((producer =semget(ftok("producer",0),1,IPC_CREAT|0660))==-1)
18    {
19          printf("semget failed.\n");
20          exit(1);
21    }
22    init_a_semaphore(producer,0,1);
23    /*获得并连接名为“shared”的共享内存*/
24    if((shmid=shmget(ftok("shared",0),SHMSZ, 0666|IPC_CREAT))==-1)
25    {
26          printf("shmget failed.\n");
27          exit(1);
28    }
29    if((shm=shmat(shmid,(unsigned char *)0,0))==-1)
30    {
31          printf("shmat failed.\n");
32          exit(1);
33    }
34    /*从共享内存中读服务进程所写数据，并输出它们 */
35    for(i=0; ; i++)
36    {
37          semaphore_V(consumer);    /* 让服务进程生成数据 */
38          semaphore_P(producer);    /* 等待数据生成完毕 */
39          printf("Data receiverd: %s\n",shm);
40          /*处理数据，遇到“end”结束循环*/
41          sleep(1);
42          if(strcmp(shm,"end")==0)
43          break;
44    }
45    semctl(producer,0,IPC_RMID,0);       /*删除信号量 producer*/
46    semctl(consumer,0,IPC_RMID,0);       /*删除信号量 consumer*/
47  }
```

说明：数据处理程序在一个循环中运行(第 35~44 行)。每一个循环迭代处理数据生成程序一次写入的数据。为此，数据处理程序首先对信号量 consumer 执行 V 操作(第 37 行)，指出共享内存缓冲是自由的，这使得数据生成程序能够写共享内存缓冲；然后，对 producer

信号量执行 P 操作(第 38 行)，等待数据生成程序在共享内存缓冲中产生数据。当 P 操作完成时，数据处理程序知道有某些数据正在共享内存缓冲中等待处理，于是开始处理这批数据。所做的处理只是简单地打印这些数据(第 39 行)。当数据处理完毕时，它开始循环的下一个迭代。这一过程直到数据处理程序收到一个 end 字符串为止(第 42~43 行)。

下面是数据产生程序

```
1   /* ex21 server.c*/
2   #include     "myshm.h"
3   int main()
4   {
5     char *shm, *s;
6     int shmid;
7     int producer, consumer,i;
8     char readbuf[SHMSZ];
9     /*创建和初始化信号量 consumer */
10    if((consumer =semget(ftok("consumer",0),1,IPC_CREAT|0660))==-1)
11    {
12            printf("semget failed.\n");
13            exit(1);
14    }
15    init_a_semaphore(consumer,0,1);
16    /*创建和初始化信号量 producer */
17    if((producer =semget(ftok("producer",0),1,IPC_CREAT|0660))==-1)
18    {
19            printf("semget failed.\n");
20            exit(1);
21    }
22    init_a_semaphore(producer,0,1);
23    /*获得并连接名为“shared”的共享内存*/
24    if((shmid=shmget(ftok("shared",0),SHMSZ, 0666|IPC_CREAT))==-1)
25    {
26            printf("shmget failed.\n");
27            exit(1);
28    }
29    if((shm=shmat(shmid,(unsigned char *)0,0))==-1)
30    {
31            printf("shmat failed.\n");
32            exit(1);
33    }
34    /*从标准输入读数据并写到共享内存*/
35    for(i=0; ; i++)
36    {
37            /*读入数据*/
```

```
38              printf("Enter text:");
39              fgets(readbuf,SHMSZ,stdin);
40              /*  等待客户进程释放共享内存  */
41              semaphore_P(consumer);
42              sprintf(shm,"Message %4d from producer %d is \"%s\" \n",i, getpid(),readbuf);
43              semaphore_V(producer);     /*通知客户进程取数据*/
44              if(strcmp(readbuf,"end")==0)
45                      break;
46      }
47      return 0;
48  }
```

数据生成程序也在一个循环中运行(第 35~46 行)。每一个循环迭代在共享内存缓冲中生成一批数据，这批数据是从标准输入读入的一行字符串。(第 39 行)它首先强制对信号量 consumer 执行 P 操作(第 41 行)，以等待共享存储缓冲是自由的；当 P 操作完成时，它往共享内存缓冲中写入数据，写入的数据来自终端输入。一旦数据写完，它便对 producer 信号量执行 V 操作(第 43 行)，告诉数据处理程序在共享内存缓冲中有待处理的数据。如果读入的数据是 end，程序跳出循环结束执行，否则继续生成下一批数据(第 44~45 行)。

6.5 小　　结

本章详细地介绍了 Linux 操作系统中最有特点的一个方面，即进程间通信(IPC)。Linux 支持传统的 Unix 信号和管道机制，而且还支持 System V IPC 的机制，包括消息队列、信号量、共享内存。本章介绍了这几种 IPC 机制的概念、特点和使用方法，并详细讲解创建、控制、删除等操作的相关函数调用。完成本章的学习后，应该掌握了有关本地进程通信程序的编写方法。

习　　题

一、填空题

1. 进程间通信有如下一些目的：______、______、______、______、______。

2. Linux 支持 Unix System v 中的三种进程间通信机制，它们是：______、______、______。

3. 在实际应用中，一个用户进程常常需要对多个信号作出处理。为了方便对多信号进

行处理，在 Linux 系统中引入______的概念。

4. 命名管道又叫______。

5. 每一个 System V IPC 资源有 2 个唯一的标志与之相连，即______和______。

6. 消息队列是一条由消息连接而成的______，它保存在内核中，通过消息队列的______来访问。

7. 信号量实际上是个______，主要用来控制多个进程对______的访问。

8. ______机制是最快的一种进程通信机制。

二、选择题

1. 进程可以忽略大部分信号，但下列信号中______是不能忽略的。

(A) SIGHUP (B) SIGINT (C) SIGSTOP (D) SIGQUIT

2. 在 kill(pid, signum)函数中，pid 参数表示 kill 函数发送信号对象的进程号或进程组号。pid>0 表示______。

(A) 向进程号为 pid 值的进程发送信号

(B) 向与发送信号的进程有相同进程组号的进程发送信号

(C) 向进程组号为 pid 绝对值的进程组发送信号

(D) 未定义

3. 可以使用______命令得到 IPC 机制中所有对象的状态。

(A)ls (B) cd (C) kill (D) ipcs

4. 打开或创建消息队列的函数是______。

(A)msgget (B) msginit (C) msgcreate (D) msg

5. 创建或打开信号量集的系统函数是______。

(A) seminit (B) sem (C) semget (D) semcreate

6. 创建共享内存的函数是______。

(A) shminit (B) seminit (C) semget (D) shmget

三、上机题

1. 编写一个程序，将用户从命令行指定的进程用 SIGTERM 信号或指定的信号结束。

2. 编写一个多客户-单一服务器模式的程序，用命名管道实现客户到服务器之间传递数据的操作。

3. 编写用消息队列进行通信的程序，其中一个进程负责读入文本文件的内容，另一个进程负责显示读取的内容。

4. 用共享内存的方法重新编写例 6-3 的程序。

CHAPTER 7

线 程 操 作

线程(thread)技术早在 60 年代就被提出，但真正应用多线程到操作系统中去，是在 80 年代中期，Solaris 是这方面的佼佼者。传统的 UNIX 也支持线程的概念，但是在一个进程(process)中只允许有一个线程，这样多线程就意味着多进程。现在，多线程技术已经被许多操作系统所支持，包括 Windows，当然，也包括 Linux。

多线程程序作为一种多任务、并发的工作方式，有以下的优点：

(1) 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作(time consuming)置于一个新的线程，可以避免这种尴尬的情况。

(2) 使多CPU系统更加有效。操作系统会保证当线程数不大于 CPU 数目时，不同的线程运行于不同的 CPU 上。

(3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

本章介绍一下与 Linux 线程编程相关的内容。

7.1 线 程 概 述

许多程序必须执行一些独立的不需要串行化的任务。比如，一个数据库服务器应该能监听和处理大量的客户请求。因为这些请求不需要按照一个特定的顺序来得到服务，所以它们可以被当作独立的执行体来看待，原则上它们可以是并行运行的。如果系统提供了子任务可以并发执行的机制，那么这些应用程序可以执行得更好。

在传统的 UNIX 系统中，这些程序使用多个进程。许多关键的服务器应用程序有一个

监听进程在不停地运行，等待客户请求到来。当一个请求到达时，这个监听进程创建(fork)一个新的进程为这个请求服务。因为对请求进行服务经常包括一些 I/O 操作，它可能阻塞进程。

在一个应用程序中使用多个进程有着一些明显的缺点。创建这些进程增加了一些基本的开销，因为 fork 是一个花销很大的系统调用。由于每个进程都有它自己的地址空间，它必须使用进程间通信的手段如消息传递或者共享内存。要把这些进程分配到不同的机器或处理器上去运行，以及在进程之间传递信息、等待进程的完成、收集结果等都需要额外的开销。

这些都说明了进程抽象概念的不充分之处，现在能够建立一个独立的计算单元的概念模型，这些计算单元是一个应用程序全部处理工作的一个部分。这些单元之间的交互相对是很少的，因此需要很少的同步。一个应用程序可能包含一个或多个这种单元。线程这种抽象概念就代表了一个单独计算单元。传统的 UNIX 进程是单线程的，这意味着所有的计算都被串行化在同一个单元之中了。

7.1.1 线程的基本概念

一个进程是一个复合的实体，可以分为两个部分：线程的集合和资源集合。线程是一个动态的对象，它表示进程中的一个控制点，并且执一系列的指令。资源包括地址空间、打开的文件、用户凭证和配额等，这些资源为进程中所有线程所共享。此外每一个线程有它自己私有对象，比如程序计数器、堆栈和寄存器的值。传统的 UNIX 进程有一个单独的控制线程，在多线程系统中进行了扩展，允许在一个进程中有多于一个的控制线程。

7.1.2 用户态线程与内核态线程

用户态线程在管理上不需要内核的参与，所以通常又称为“协作式多任务”，在进程内的这些线程统一由用户程序来切换，所以每一个线程在执行完任务后，调用任务切换功能，并向其发送信号，任务切换完成。线程对 CPU 资源的占用也切换到其他线程。通常，用户态线程在线程切换时要比内核线程的速度快，不过在几个比较成功的内核态线程库中，线程切换的速度也相当快。虽然用户态线程有许多灵活性和快速的特性，但是也存在一个严重的问题，即进程中的一个线程可能独占整个时间片，导致其他线程得不到 CPU 时间而无法运行，例如，当一个线程由于磁盘 I/O 而阻塞时，其他线程同样也不能运行。另外，用户态线程不能发挥多 CPU 机器(SMP)的性能。

内核态线程是由内核来管理的，在每一个时间片内，内核负责调度进程内的线程。由于内核参与了用户态进程的调度，所以就涉及了内核态与用户态上下文的切换。通常所说的内核态线程切换速度慢就是由于这个原因导致的。但是使用内核态线程的一个明显好处是进程内的一个线程不会独占整个进程的 CPU 时间，这样，如果一个线程由于磁盘 I/O 而

阻塞，其他线程仍可以利用 CPU 时间运行。使用核心态线程的另外一个好处是可以充分发挥 SMP 系统的性能，而且随着系统 CPU 数量的增多，应用程序运行的速度明显加快。

现在有一些线程库既支持用户态线程，也支持内核态线程。因为几个比较成功的内核态线程库在任务切换上比较出色，所以似乎没有使用用户态线程的必要。如果要使用用户态线程，有一个好处是可以在一个进程的多个线程间方便地协调任务。

7.2 线 程 管 理

典型的线程包含一个运行时间系统，它可以按透明的方式来管理线程。通常线程包括对线程的创建和删除，以及对互斥和条件变量的调用。 POSIX 标准线程库具有这些调用。这些包还提供线程的动态创建和删除，因此直到运行时间之前，线程的个数不必知道。

线程具有一个 ID、一个堆栈、一个执行优先权，以及执行的开始地址，POSIX 线程通过 pthread_t 类型的 ID 来引用。pthread_t 其实就是无符号长整数，在文件/usr/include/bit/pthreadtypes.h 中有如下的定义：

```
typedef unsigned long int pthread_t;
```

线程的内部数据结构也包含调度和使用信息。进程的线程共享进程的完整地址空间，它们能够修改全局变量，访问打开的文件描述符或用别的方式相互作用。

7.2.1 创建线程和结束线程

如果线程可在进程的执行期间的任意时刻被创建，并且线程的数量事先没有必要指定，这样的线程称为动态线程。在 POSIX 中，线程是用 pthread_create 动态地创建的。pthread_create 能创建线程，并将它放入就绪队列。该函数的定义如下所示：

```
#include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

pthread_create 创建一个线程，这个线程与创建它的线程同步执行。新创建的线程将执行函数 start_routine，这个函数的参数由指针 arg 指定。这个线程可以通过 pthread_exit 来终止，或者当函数 start_routine 返回时自然终止。参数 attr 指定新线程的属性。我们将在讲述 pthread_attr_init 时具体介绍这个参数。参数 attr 可以为 NULL，此时将使用默认的属性，即使用最小的堆栈空间，使用通常的调度策略等，在后面我们将使用其他的属性，那时，读者会看到一些更有趣的应用。

与创建进程的系统调用 fork 不同的是，用 pthread_create 创建的线程并不是与父进程在同一点开始运行，而是从 pthread_create 指定的函数开始运行。这一点很明显，因为如果

线程也像子进程一样与父进程从同一点开始运行，那么将有多个线程使用同一资源。正如前面讲过的，每一个进程部有自己的地址空间和资源，而多个线程要使用一个地址空间和资源。

如果执行成功，那么将返回 0，并将新创建线程的标识符存放在由指针 thread 指向的地址。如果执行失败，那么将返回一个非零值。

在创建线程后，可以调用 pthread_self 函数得到线程的 ID，该函数的定义如下：

```
#include <pthread.h>
pthread_t pthread_self();
```

要结束一个线程，需要调用函数 pthread_exit。它的原型如下：

```
#include <pthread.h>
void pthread_exit(void *retval);
```

此函数用于结束一个线程。它将调用用户为线程注册的清除处理函数，然后结束当前线程，返回值为 retval。注册清除处理函数的方法将在后面讲到。这个函数在我们想在线程的中途退出时有用。

函数 pthread_exit 在成功调用时，返回 0，失败时返回-1。

下面看一个简单的例子。

例 7-1 程序中将使用两个线程，一个打印自己的线程 ID 和 Hello，另一个打印自己的线程 ID 号和 World。

```
1   /* ex1.c */
2   #include <stddef.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <pthread.h>
6
7   void print_message(char*ptr);
8
9   int main()
10  {
11    pthread_t thread1, thread2;
12    char *msg1="Hello\n";
13    char *msg2="World\n";
14    pthread_create(&thread1,NULL, (void *)(&print_message), (void *)msg1);
15    pthread_create(&thread2,NULL, (void *)(&print_message), (void *)msg2);
16    sleep(1);
17    return 0;
18  }
19
20  void print_message(char *ptr)
```

```
21 {
22  int retval;
23  printf("Thread ID: %lx", pthread_slef());
24  printf("%s",ptr);
25  pthread_exit(&retval);
26 }
```

说明：程序用 pthread_create 函数创建了 2 个线程(第 14、15 行)，线程函数是 print_message 函数(第 20~26 行)，第一个线程使用 msg1 作为函数 print_message 的参数，在打印了自己的线程 ID 和 Hello 之后，调用 pthread_exit 函数退出(第 23~25 行)，第二个线程也做同样处理。

用 gcc 编译多线程程序时，必须与 pthread 函数库连接。下面的方法可以做到这一点：

```
gcc –lpthread –o ex1 ex1.c
```

上述编译命令把程序与 pthread 函数库相连，输出可执行文件为 ex1。下面是可能的程序执行结果：

```
./ex1
Thread ID: b7d86b90
Hello
Thread ID: b7585b90
World
```

虽然这个程序很简单，但是有两个明显的缺陷：

(1) 因为线程是同步执行的，所以无法保证第一个线程先打印，而第二个线程后打印。所以输出结果可能是 Hello World，也可能是 World Hello。

(2) 如果在两个线程打印之前，主程序执行到了 return 那么将没有打印结果输出。因为在主程序执行 return 后，所有的线程都要终止执行。因此在程序中加入了 sleep 函数(第 16 行)，让主线程休眠 1 秒钟，等新创建的 2 个线程输出信息后再继续执行。

上面的程序虽然很短，但是也涉及到了多线程编程中经常会遇到的问题，即竞争问题。例如两个线程要竞争看哪一个线程会先调用 print_message 函数，为了让线程输出正确的结果，必须引入线程的同步机制。后面的内容会讨论到这个问题。

7.2.2 挂起线程

可以使用下面的函数将一个线程挂起：

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

此函数用于挂起当前线程直至指定线程终止。参数 th 是一个线程标识符，用于指定要

等待其终止的线程。参数 thread_return 用于存放其他线程的返回值。对于每一个可连接的线程都必须调用该函数一次。任何线程都不能对相同的线程调用此函数。

如果执行成功，那么参数 th 的返回值将保存在由参数 thread_return 指向的地址中，函数返回 0，否则返回一个非零值。

下面的程序说明了 pthread_join 函数的使用。

例 7-2 pthread_join 函数的使用。

```
1  /*ex2.c */
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <pthread.h>
6
7  void print_msg(char *ptr);
8
9  int main()
10 {
11   pthread_t thread1, thread2;
12   int i,j;
13   void *retval;
14   char *msg1="Hello\n";
15   char *msg2="World\n";
16   pthread_create(&thread1,NULL, (void *)(&print_msg), (void *)msg1);
17   pthread_create(&thread2,NULL, (void *)(&print_msg), (void *)msg2);
18   pthread_join(thread1,&retval);
19   pthread_join(thread2,&retval);
20   return 0;
21 }
22
23 void   print_msg(char *ptr)
24 {
25   int i;
26   for(i=0;i<10000;i++)
27           printf("%s",ptr);
28 }
```

说明：程序 2 与程序 1 功能基本相同，不同的是线程函数 print_msg 是要打印输出 10000 次(第 23~28 行)，同时在程序中加入了 pthread_join 函数等待 2 个线程结束(第 18~19 行)，这样可以保证这两个线程在完成工作后，主进程才能退出。

7.2.3　线程同步

当在同一内存空间运行多个线程时，要注意一个基本的问题是不要让线程之间互相破坏。例如，两个线程要更新两个变量的值。一个线程要把两个变量的值都设成 0，另一个线程要把两个变量的值都设成 1。如果两个线程同时要做这件事情，结果可能是，一个变量的值是 0，另一个变量的值是 1。这是因为正好在第一个线程把第一个变量设为 0 后，运行环境切换，第二个线程将把两个变量都设成 1，然后运行环境再切换，第一个线程恢复运行，把第二个变量设成 0。结果就是，一个变量的值是 0、另一个变量的值是 1。

按照 POSIX 标准，POSIX 提供了两种类型的同步机制，它们是互斥锁(mutex)和条件变量(condition variable)。互斥锁是一个简单的锁定命令，它可以用来锁定对共享资源的访问。对于线程来说，整个地址空间都是共享的资源，所以线程的任何资源都是共享的资源。互斥锁的特点是：

(1) 原子性：把一个互斥锁定为一个原子操作，这意味着操作系统(或 pthread 函数库)保证了如果一个线程锁定了一个互斥锁，没有其他线程在同一时间可以成功锁定这个互斥锁。

(2) 唯一性：如果一个线程锁定了一个互斥锁、在它解除锁定之前，没有其他线程可以锁定这个互斥量。

(3) 非繁忙等待：如果一个线程已经锁定了一个互斥锁，第二个线程又试图去锁定这个互斥锁，则第二个线程将被挂起(不占用任何 CPIJ 资源)，直到第一个线程解除对这个互斥锁的锁定为止，第二个线程则被唤醒并继续执行，同时锁定这个互斥锁。

下面几个函数是处理互斥锁时常用的几个函数。

1. pthread_mutex_init 函数

```
#include <pthread.h>
pthread_mutex_t fastmutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex=PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex=PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr *attr);
```

上面三个常量是常用的处理互斥锁的常量。

pthread_mutex_init 用来初始化一个由参数 mutex 指向的互斥锁，这个互斥锁的属性由参数 attr 指定，或者通过指定 attr 为 NULL 而使用默认的属性。

不会出现有多个线程同时初始化同一个互斥锁的情形，一个互斥锁在使用期间一定不会被重新初始化。

如果 pthread_mutex_init 执行成功，则返回 0，并将新创建的互斥锁的 ID 值放到参数 mutex 中。如果执行失败，那么将返回一个错误编号。

2. pthread_mutex_destroy 函数

函数原型：

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

用 pthread_mutex_destroy 函数解除由参数 mutex 指向的互斥锁的任何状态。储存互斥锁的内存并不被释放。

如果 pthread_mutex_destroy 执行成功则返回 0；如果执行失败，那么将返回一个错误编号。

3. pthread_mutex_lock 函数

函数原型：

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

用 pthread_mutex_lock 函数可以锁定由参数 mutex 指向的互斥锁。如果 mutex 已经被锁定，那么当前调用的线程将阻塞直到互斥锁被其他线程释放(阻塞线程按照线程优先级等待)。当 pthread_mutex_lock 返回时，说明互斥锁已经被当前线程成功加锁。

如果 pthread_mutex_lock 执行成功则返回 0，其他的值说明发生了错误。

4. pthread_mutex_trylock 函数

函数原型：

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

用 pthread_mutex_trylock 来尝试给由参数 mutex 指定的互斥锁加锁。这个函数是 pthread_mutex_lock 的非阻塞版本。pthread_mutex_lock 在给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_lock 将一直阻塞，不会立即返回。而使用 pthread_mutex_trylock 给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_trylock 调用将返回错误。否则，互斥锁将被调用者加锁。

如果 pthread_mutex_trylock 执行成功则返回 0，其他值意味着错误。

5. pthread_mutex_unlock 函数

函数原型：

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

用 pthread_mutex_unlock 给由参数 mutex 指定的互斥锁解锁。互斥锁必须处于加锁状态而且调用本函数的线程必须是给互斥锁加锁的同一个线程才能给互斥锁解锁。如果有其他线程在等待互斥锁，那么有核心的调度程序决定哪个线程将获得互斥锁并脱离阻塞状态。

如果 pthread_mutex_unlock 执行成功，则返回 0。其他值意味着错误。

下面我们介绍一个简单的读/写程序，在这个程序中，一个线程从共享的缓冲区中读数据，另一个线程向共享的缓冲区中写数据。对共享的缓冲区的访问控制是通过使用一个互斥锁来实现的。

例 7-3　线程同步的例子。

```
1   /*ex3.c*/
2
3   #include <stddef.h>
4   #include <stdio.h>
5   #include <unistd.h>
6   #include <pthread.h>
7
8   #define FALSE 0
9   #define TRUE 1
10
11  void readfun();
12  void writefun();
13
14  char buffer[256];
15  int buffer_has_item=0;
16  int retflag=FALSE;
17  pthread_mutex_t mutex;
18
19  int main()
20  {
21    pthread_t reader;
22    pthread_mutex_init(&mutex,NULL);
23    pthread_create(&reader,NULL,(void *)&readfun,NULL);
24    writefun();
25  }
26
27  void readfun()
28  {
29    while(1)
30    {
31            if(retflag)
32                    return;
33            pthread_mutex_lock(&mutex);
34            if(buffer_has_item==1)
35            {
36                    printf("%s",buffer);
37                    buffer_has_item=0;
```

```
38            }
39            pthread_mutex_unlock(&mutex);
40    }
41  }
42  void writefun()
43  {
44    int i=0;
45    while(1)
46    {
47            if(i==10)
48            {
49                    retflag=TRUE;
50                    return;
51            }
52            pthread_mutex_lock(&mutex);
53            if(buffer_has_item==0)
54            {
55                    sprintf(buffer,"This is %d\n",i++);
56                    buffer_has_item=1;
57            }
58    pthread_mutex_unlock(&mutex);
59    }
60  }
```

说明：在上面的程序中定义了一个互斥锁 mutex(第 17 行)，在主函数 main 中，首先初始化互斥锁(第 22 行)，然后创建读线程(第 23 行)，向共享缓冲区中写数据(第 24 行)。假定缓冲区只有两种状态：有数据和无数据。写线程首先锁定互斥锁(第 52 行)，然后检查缓冲区是否为空。如果缓冲区为空，那么就向缓冲区中写数据(第 55 行)，并将标记 buffer_has_item 置为 1(第 56 行)，这样读线程就知道缓冲区中有数据了。写完数据后，将互斥锁解锁(第 58 行)。

读线程与写线程的过程类似，首先锁定互斥锁，然后检查缓冲区中是否有数据，如果有就把数据取出来打印输出，然后释放对互斥锁的锁定(第 33~39 行)。

程序执行结果为：

```
./ex3
This is 0
This is 1
This is 2
This is 3
This is 4
This is 5
This is 6
```

```
This is 7
This is 8
This is 9
```

如果程序中不再需要互斥锁，那么可以调用 pthread_mutex_destroy(&mutex)来清除互斥锁。

在初始化互斥锁时，我们使用了默认的属性 NULL。

在程序中使用互斥锁虽然可以解决一些资源竞争的问题，但是互斥锁只有两种状态，这使得它的用途非常有限。

在 POSIX 中还提供了另外一种同步机制，即条件变量。条件变量是对互斥锁的补充，它允许线程阻塞并等待另一个线程发送的信号。当收到信号时，阻塞的线程就被唤醒并试图锁定与之相关的互斥锁。

下面是 Linux 的线程库中处理条件变量的一些函数。

6. pthread_cond_init 函数

函数原型：

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const pthread_cond_attr *attr);
```

用 pthread_cond_init 初始化由参数 cond 指定的条件变量。这个条件变量的属性由参数 attr 指定。如果参数 attr 为 NULL。那么就使用默认的属性设置。

多线程不能同时初始化同一个条件变量。如果一个条件变量正在使用，它不能被重新初始化。

如果 pthread_cond_init 执行成功，则返回 0，并将新创建的条件变量的 ID 放在参数 cond 中，如果返回其他的值意味着有错误。

7. pthread_cond_destroy 函数

函数原型：

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

使用 pthread_cond_destroy 来清除由参数 cond 指向的条件变量的任何状态。但是储存条件变量的内存空间不被释放。

如果函数 pthread_cond_destroy 执行成功则返回 0，其他值意味着错误。

8. pthread_cond_wait 函数

函数原型：

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

使用 pthread_cond_wait 释放由参数 mutex 指向的互斥锁，并且使调用线程关于参数 cond 指向的条件变量阻塞。被阻塞的线程可以被 pthread_cond_signal、pthread_cond_broadcast 或者由 fork 和传递信号引起的中断唤醒。

即使返回错误信息，pthread_cond_wait 通常在互斥锁被调用线程加锁后才返回。

函数将阻塞直到条件变量被信号唤醒。它在阻塞前自动释放互斥锁，在返回前再自动获得它。

如果有多个线程关于条件变量阻塞，其退出阻塞状态的顺序将不确定。

如果 ptread_cond_wait 执行成功则返回 0。其他值意味着错误。

9. pthread_cond_timewait 函数

函数原型：

```
#include <pthread.h>
int pthread_cond_timewait(pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime);
```

pthread_cond_timedwait 和 pthread_cond_wait 的用法相似，区别在于 pthread_cond_timedwait 在经过由参数 abstime 指定的时间时不阻塞。

即使是返回错误，pthread_cond_timedwait 也只在给互斥锁加锁后返回。

pthread_cond_timedwait 函数将阻塞，直到条件变量获得信号或者经过由 abstime 指定的时间。

如果 pthread_cond_timedwait 执行成功则返回零。如果阻塞条件变量的时间超过了由参数 abstime 所指定的时间，那么就返回 ETIMEOUT。其他值意味着错误。

10. pthread_cond_signal 函数

函数原型：

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

使用 pthread_cond_signal 使得关于由参数 cond 指向的条件变量阻塞的线程退出阻塞状态。在同一个互斥锁的保护下使用 pthread_cond_signal，否则，条件变量可以在对关联条件变量的测试和 pthread_cond_wait 带来的阻塞之间获得信号，这将导致无限期的等待。

如果没有一个线程关于条件变量阻塞，那么 pthread_cond_signal 无效。

如果 pthread_cond_signal 执行成功则返回 0。其他值意味着错误。

11. pthread_cond_broadcast 函数

函数原型：

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

使用 pthread_cond_broadcast 使得所有关于由参数 cond 指向的条件变量阻塞的线程退出阻塞状态。如果没有阻塞的线程，cond_broadcast 无效。

这个函数将唤醒所有由 pthread_cond_wait 阻塞的线程。因为所有关于条件变量阻塞的线程都同时参与竞争，所以使用这个函数需要小心。

如果 pthread_cond_broadcast 执行成功则返回 0。其他值意味着错误。

下面通过一个例子来看一看如何使用 Linux 的线程库来处理互斥锁和条件变量。

例 7-4 这个例子是 Linux 的线程库带的一个示范程序。这个程序是一个典型的生产者/消费者的例子。

```
1   /* ex4.c */
2   #include <stdio.h>
3   #include <pthread.h>
4   #define BUFFER_SIZE 4
5   #define OVER (-1)
6   struct producers                          //定义生产者条件变量结构
7   {
8     int buffer[BUFFER_SIZE];                //定义缓冲区
9     pthread_mutex_t lock;                   //定义访问缓冲区的互斥锁
10    int readpos, writepos;                  //读写的位置
11    pthread_cond_t   notempty;              //缓冲区有数据时的标记
12    pthread_cond_t   notfull;               //缓冲区未满的标记
13  };
14  //初始化缓冲区
15  void init(struct producers *b)
16  {
17    pthread_mutex_init(&b->lock,NULL);
18    pthread_cond_init(&b->notempty,NULL);
19    pthread_cond_init(&b->notfull,NULL);
20    b->readpos=0;
21    b->writepos=0;
22  }
23  //在缓冲区中存放一个整数。
24  void put(struct producers *b, int data)
25  {
26    pthread_mutex_lock(&b->lock);
27    //当缓冲区为满时等待。
28    while((b->writepos+1)%BUFFER_SIZE==b->readpos)
29    {
30            pthread_cond_wait(&b->notfull,&b->lock);
31     //在返回之前，pthread_cond_wait 需要参数 b->lock。
32    }
33    //向缓冲区中写数据，并将写指针向前移动。
```

```
34    b->buffer[b->writepos]=data;
35    b->writepos++;
36    if(b->writepos>=BUFFER_SIZE) b->writepos=0;
37    //发送当前缓冲区中有数据的信号。
38    pthread_cond_signal(&b->notempty);
39    pthread_mutex_unlock(&b->lock);
40  }
41  //从缓冲区中读数据并将数据从缓冲区中移走。
42  int get(struct producers *b)
43  {
44    int data;
45    pthread_mutex_lock(&b->lock);
46    //当缓冲区中有数据时等待。
47    while(b->writepos==b->readpos)
48    {
49            pthread_cond_wait(&b->notempty,&b->lock);
50    }
51    //从缓冲区中读数据，并将指针前移。
52    data=b->buffer[b->readpos];
53    b->readpos++;
54    if(b->readpos>=BUFFER_SIZE) b->readpos=0;
55    //发送当前缓冲区未满的信号。
56    pthread_cond_signal(&b->notfull);
57    pthread_mutex_unlock(&b->lock);
58    return data;
59  }
60
61  struct producers    buffer;
62  void *producer(void *data)
63  {
64    int n;
65    for(n=0;n<10;n++)
66    {
67            printf("Producer: %d-->\n",n);
68            put(&buffer,n);
69    }
70    put(&buffer,OVER);
71    return NULL;
72  }
73
74  void *consumer(void *data)
75  {
76    int d;
```

```
77    while(1)
78    {
79          d=get(&buffer);
80          if(d==OVER) break;
81          printf("Consumer: --> %d\n",d);
82    }
83    return NULL;
84  }
85
86  int main()
87  {
88    pthread_t tha,thb;
89    void *retval;
90    init(&buffer);
91    pthread_create(&tha,NULL,producer,0);
92    pthread_create(&thb,NULL,consumer,0);
93    pthread_join(tha,&retval);
94    pthread_join(thb,&retval);
95    return 0;
96
97  }
```

说明：主进程创建两个线程，一个叫做 producer(第 91 行)，另外一个叫做 consumer(第 92 行)。producer 向缓冲区中写整数 1~10000(第 62~72 行)，当缓冲区中已经写入数据后，就发送缓冲区中有数据的信号(第 24~40 行)。consumer 从缓冲区中读数据(第 74~84 行)，当 consumer 从缓冲区中读出数据后，就发送当前缓冲区未满的信号(第 42~59 行)。程序中其他语句的功能已在注释中说明。

程序执行结果为：

```
$ ./ex4
Producer : 0-->
Producer : 1-->
Producer : 2-->
Producer : 3-->
Consumer: --> 0
Consumer: --> 1
Consumer: --> 2
Producer : 4-->
Producer : 5-->
Producer : 6-->
Consumer: --> 3
Consumer: --> 4
Consumer: --> 5
```

```
Producer : 7-->
Producer : 8-->
Producer : 9-->
Consumer: --> 6
Consumer: --> 7
Consumer: --> 8
Consumer: --> 9
```

从上面的程序中，我们可以看出利用条件变量可以更加灵活地在线程间通信，使线程更加方便地加以控制资源。

除了互斥锁和条件变量来同步线程的执行外，也可以通过信号量来同步线程的执行，下面还是通过制造者和消费者的例子来说明怎样通过信号量来同步线程的执行。

例 7-5 使用信号量同步线程的执行。

```
1   /* ex5.c*/
2   #include <stdio.h>
3   #include <pthread.h>
4   #include <semaphore.h>
5   #define BUFFER_SIZE 4
6   #define OVER (-1)
7   struct producers
8   {
9     int buffer[BUFFER_SIZE];
10    int readpos, writepos;
11    sem_t sem_read;
12    sem_t sem_write;
13  };
14
15  void init(struct producers *b)
16  {
17    sem_init(&b->sem_write,0,BUFFER_SIZE-1);
18    sem_init(&b->sem_read,0,0);
19    b->readpos=0;
20    b->writepos=0;
21  }
22
23  void put(struct producers *b, int data)
24  {
25    sem_wait(&b->sem_write);
26    b->buffer[b->writepos]=data;
27    b->writepos++;
28    if(b->writepos>=BUFFER_SIZE) b->writepos=0;
29    sem_post(&b->sem_read);
```

```
30 }
31
32 int get(struct producers *b)
33 {
34   int data;
35   sem_wait(&b->sem_read);
36   data=b->buffer[b->readpos];
37   b->readpos++;
38   if(b->readpos>=BUFFER_SIZE) b->readpos=0;
39   sem_post(&b->sem_write);
40   return data;
41 }
42
43 struct producers    buffer;
44 void *producer(void *data)
45 {
46   int n;
47   for(n=0;n<10;n++)
48   {
49           printf("Producer : %d-->\n",n);
50           put(&buffer,n);
51   }
52   put(&buffer,OVER);
53   return NULL;
54 }
55
56 void *consumer(void *data)
57 {
58   int d;
59   while(1)
60   {
61           d=get(&buffer);
62           if(d==OVER) break;
63           printf("Consumer: --> %d\n",d);
64   }
65   return NULL;
66 }
67
68 int main()
69 {
70   pthread_t tha,thb;
71   void *retval;
72   init(&buffer);
```

```
73    pthread_create(&tha,NULL,producer,0);
74    pthread_create(&thb,NULL,consumer,0);
75    pthread_join(tha,&retval);
76    pthread_join(thb,&retval);
77    return 0;
78    }
```

说明： 上面的程序与例 7-4 基本相同，不同是在结构 producers 中用信号量 sem_read 和 sem_write 取代了互斥锁和条件变量(第 11~12 行)，相应的初始化函数中也该为信号量初始化函数(第 17~18 行)。在 put 函数中，首先等待 sem_write 信号量有效，之后开始往缓冲区中填充数据，缓冲区满后，设置 sem_read 信号量有效(第 23~30 行)。在 get 函数中，等待 sem_read 信号量有效，之后开始读取缓冲区，全部数据读取完毕后，设置 sem_wrtie 信号量有效(第 32~41 行)。程序的其他部分与例 7-4 相同，此处不再赘述。程序的执行结果为：

```
$ ./ex5
Producer : 0-->
Producer : 1-->
Producer : 2-->
Producer : 3-->
Consumer: --> 0
Consumer: --> 1
Consumer: --> 2
Producer : 4-->
Producer : 5-->
Producer : 6-->
Consumer: --> 3
Consumer: --> 4
Consumer: --> 5
Producer : 7-->
Producer : 8-->
Producer : 9-->
Consumer: --> 6
Consumer: --> 7
Consumer: --> 8
Consumer: --> 9
```

从执行结果可以看出，例 7-5 实现的功能同例 7-4 相同，都正确地实现了线程间的同步。

7.2.4 取消线程和取消处理程序

用户可以调用函数来取消一个线程。相应的函数说明如下。但要注意这种操作不是用于取消当前线程，而是当前线程调用函数来取消另一个线程。

Linux 的线程库中用于取消线程的几个函数包括：

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel();
```

Linux 允许一个线程终止另外一个线程的执行，即一个线程可以向另外一个线程发出一个终止请求。根据不同的设置，接收到这个终止请求的线程可以忽略这个请求，也可以立即终止或者延长一段时间后终止。

当一个线程最终响应终止请求时，它所执行的操作与调用 pthread_exit 函数时一样，即调用每一个 cleanup 处理程序，然后调用与线程相关的数据处理程序，最后线程终止执行。

- 函数 pthread_cancel 发送一个终止请求到由参数 thread 指定的线程。调用成功时，返回值为 0；调用失败时，返回错误代码。
- 函数 pthread_setcancelstate 用于设置调用函数的线程自身的状态。参数 state 是要设置的新的状态。参数 oldstate 是指向存放要设置的状态的缓冲区的指针。调用成功时，返回值为 0；调用失败时，返回错误代码。
- 函数 pthread_setcanceltype 用于设置对取消的响应方式。响应方式有两种。
  PTHREAD_CANCEL_ASYNCHRONOUS：立刻取消。
  PTHREAD_CANCEL_DEFERRED：延迟取消至取消点。
  参数 type 是要设置的新的方式。参数 oldtype 是指向存放要设置的方式的缓冲区的指针。调用成功时，返回值为 0；调用失败时，返回错误代码。
- 函数 ptbread_testcancel 用于设置取消点。如果延迟取消请求挂起，那么此函数将取消当前进程。

线程终止后，要做一些清理工作，Linux 线程库提供了相应的处理函数。

1. pthread_cleanup_push 函数。

函数原型：

```
#include <pthread.h>
void pthread_cleanup_push(void(*routine) (void *), voie *arg);
```

pthread_cleanup_push 函数将子程序 routine 连同它的参数 arg 一起压入当前线程的 cleanup 处理程序的堆栈。当当前线程调用 pthread_exit 或者是通过 pthread_cancel 终止执行时，堆栈中的处理程序将按照压入堆栈时的相反顺序依次调用。

如果 pthread_cleanup_push 执行失败，则不会返回任何错误报告。

2. pthread_cleanup_pop 函数

函数原型：

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

函数 pthread_cleanup_pop 从线程的 cleanup 处理程序堆栈中弹出最上面的一个处理程序并执行它。

如果 pthread_cleanup_pop 执行失败，则不会返回任何错误报告。

这两个函数用于指明线程终止时进行什么操作。下面通过一个例子来学习线程的取消处理。

例 7-6 取消线程以及取消线程处理程序示例。

```
1   /* ex6.c*/
2   #include <stdio.h>
3   #include <unistd.h>
4   #include <stdlib.h>
5   #include <sys/types.h>
6   #include <pthread.h>
7   #include <errno.h>
8   /*定义查抄线程的数量。*/
9   #define NUM_THREADS 5
10  /*定义函数原型*/
11  void *search(void *);
12  void print_it(void *);
13  /*定义全局变量 */
14  pthread_t threads[NUM_THREADS];
15  pthread_mutex_t lock;
16  int tries;
17  int started;
18  int main(int argc, char *argv[])
19  {
20    int i,pid;
21    /* 创建一个查找的数。*/
22    pid=getpid();
23    printf("Searching for the number =%d...\n",pid);
24    /* 初始化互斥锁 */
25    pthread_mutex_init(&lock,NULL);
26    /*创建一个查找线程 */
27    for(started=0;started<NUM_THREADS;started++)
28            pthread_create(&threads[started],NULL,search,(void *)pid);
29    /*等待查找线程执行完毕*/
30    for(i=0;i<NUM_THREADS;i++)
```

```
31          pthread_join(threads[i],NULL);
32    printf("It took %d tries to find the number.\n",tries);
33    /*  退出程序  */
34    return 0;
35  }
36    /*线程终止时的清理程序*/
37  void print_it( void *arg)
38  {
39    int *try=(int *)arg;
40    pthread_t tid;
41    /*  获取调用线程的线程 ID 号  */
42    tid=pthread_self();
43    /*  打印当线程终止时的查找位置*/
44    printf("Thread %lx was canceled on its %d try.\n",tid, *try);
45  }
46  /*  线程的查找子程序  */
47  void *search(void *arg)
48  {
49    int num=(int)arg;
50    int i,j,ntries;
51    pthread_t tid;
52    /*  获取调用线程的线程 ID 号*/
53    tid=pthread_self();
54    /*  使用线程 ID 号来做随机数生成函数的种子*/
55    while(pthread_mutex_trylock(&lock)==EBUSY)
56          pthread_testcancel();
57    srand((int)tid);
58    i=rand()&0xFFFFFF;
59    pthread_mutex_unlock(&lock);
60    /*设置线程的终止参数：允许线程终止和终止延迟*/
61    ntries=0;
62    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
63    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
64    while(started<NUM_THREADS)
65          sched_yield();
66    /*将线程的清理例程压入线程清理堆栈中，这个清理例程在线程终止时调用*/
67    pthread_cleanup_push(print_it,(void *)&ntries);
68    while(1)
69    {
70          i=(i+1)&0xffffff;
71          ntries++;
72          /*检验随机数是否与目标一致*/
73          if(num==i)
```

```
74          {
75                  /*  如果互斥锁已经锁定，那么检查线程是否已经被终止。
76                  /*  如果互斥锁没有锁定，  继续下面的执行。*/
77                  while(pthread_mutex_trylock(&lock)==EBUSY)
78                          pthread_testcancel();
79                  /*设置尝试次数的全局变量  */
80                  tries=ntries;
81                  printf("Thread %lx found the number!\n",tid);
82                  /*  终止其他所有的线程*/
83                  for(j=0;j<NUM_THREADS;j++)
84                          if(threads[j]!=tid) pthread_cancel(threads[j]);
85                  break;
86          }
87          /*  每经过 100 次尝试就检查一下线程是否已经终止了*/
88          if(ntries%100==0)
89          {
90          pthread_testcancel();
91          }
92      }
93      /*  只有当线程从上面的 while 循环跳出时，才能执行下面的代码。*/
94      pthread_cleanup_pop(0);
95      return ((void *)0);
96  }
```

说明：上面程序的功能是，以主进程的 ID 号作为一个查找目标(第 22 行)，开启 5 个线程(第 27~28 行)，在查找线程中，以线程 ID 作为随机数种子，产生一个随机数作为初始值开始查找这个目标数，查找完成后，打印输出查找的次数，并结束线程，作相应的线程终止处理(第 47~96 行)。更详细的说明见程序注释。

程序执行结果为：

```
$ ./ex6
Searching for the number =7282...
Thread b7e28b90 found the number!
Thread b7627b90 was canceled on its 463200 try.
It took 5002186 tries to find the number.
```

7.2.5　线程特定数据的处理函数

因为一个进程中的线程要共享同一个地址空间，那么在这个共享空间中，所有的变量都将为所有的线程所公用。如果要使用某些一个线程特定的变量，该如何处理呢?下面先介绍一下线程特定数据的处理函数，然后通过一个例子来说明这些函数的使用方法。

1. pthread_key_create 函数

函数原型：

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void(*dest_routine(void *)));
```

函数 pthread_key_create 创建一个对进程中的所有线程都可见的关键字。这个关键字可以通过函数 pthread_setspecific 和 pthread_getspecific 来读取和设置。

当创建一个关键字时，进程中的所有线程的这个关键字的值都为 NULL，当创建一个线程时，这个线程的所有关键字的值都为 NULL。

如果 pthread_key_create 执行成功，则返回 0，并在参数 key 中保存新创建的关键字的 ID。其他的值意味着错误。

2. pthread_key_delete 函数

函数原型：

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

函数 pthread_key_delete 清除由参数 key 指定的关键字。

如果 pthread_key_delete 执行成功，则返回 0，其他的值意味着错误。

3. pthread_setspecific 函数

函数原型：

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *pointer);
```

函数 pthread_setspecific 指定由参数 pointer 指定的指针指向由参数 key 指定的关键字。每一个线程都有一个互相独立的指针，这个指针指向一个特定的关键字。

如果 pthread_setspecific 执行成功，将返回 0，其他值意味着错误。

4. pthread_getspecific 函数

函数原型：

```
#include <pthread.h>
void * pthread_getspecific(pthread_key_t key);
```

函数 pthread_getspecific 用来获取由 pthread_setspecific 设置的关键字指针。

如果 pthread_getspecific 执行成功，则返回一个指向最近一次使用 pthread_setspecific 而设定的指向线程关键字的指针。

注意：返回值可能为 NULL，此时将不能区分是否为错误。

5. pthread_once 函数

函数原型：

```
#include <pthread.h>
void * pthread_once_t once_control=PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

函数 pthread_once 的目的是保证某些初始化代码至多只能执行一次。参数 once_control 指向静态的或外部的变量，这个变量初始化为 PTHREAD_ONCE_INIT。

当第一次调用 ptbread_once 时，系统将记录已经执行了初始化，后面再调用 pthread_once 时，如果参数 once_control 相同，那么就什么也不做。

pthread_once 总是返回 0。

下面的例子说明了线程特定的数据处理函数的使用方法。下面的这个程序使用了静态变量来保存调用结果。

例 7-7 线程特定数据处理函数的使用方法。

```
1  /*ex7.c*/
2  #include <stddef.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <pthread.h>
7
8  #if 0
9  char * str_accumulate(char *s)
10 {
11   static char accu[1024]={0};
12   strcat(accu,s);
13   return accu;
14 }
15 #endif
16
17 static pthread_key_t str_key;
18 static pthread_once_t str_alloc_key_once=PTHREAD_ONCE_INIT;
19 static void str_alloc_key();
20 static void str_alloc_destroy_accu(void *accu);
21 char * str_accumulate(const char *s)
22 {
23   char *accu;
24   pthread_once(&str_alloc_key_once,str_alloc_key);
25   accu=(char *)pthread_getspecific(str_key);
26   if(accu==NULL)
27   {
28          accu=malloc(1024);
29          if(accu==NULL) return NULL;
```

```
30            accu[0]=0;
31            pthread_setspecific(str_key,(void *)accu);
32            printf("Thread %lx : allocating buffer at %p\n",pthread_self(),accu);
33     }
34   strcat (accu,s);
35   return accu;
36  }
37
38  static void str_alloc_key()
39  {
40   pthread_key_create(&str_key,str_alloc_destroy_accu);
41   printf("Thread %lx : allocated key %d\n",pthread_self(), str_key);
42  }
43
44  static void str_alloc_destroy_accu(void *accu)
45  {
46   printf("Thread %lx : freeing buffer at %p\n",pthread_self(),accu);
47   free(accu);
48  }
49
50  void *process(void *arg)
51  {
52   char *str;
53   str=str_accumulate("Result of ");
54   str=str_accumulate((char *)arg);
55   str=str_accumulate(" thread");
56   printf("Thread %lx: \"%s\" \n",pthread_self(),str);
57   return NULL;
58  }
59
60  int main(int argc, char *argv[])
61  {
62   char *str;
63   pthread_t th1,th2;
64   str=str_accumulate("Result of ");
65   pthread_create(&th1,NULL,process,(void *)"first");
66   pthread_create(&th2,NULL,process,(void *)"second");
67   str=str_accumulate("initial thread");
68   printf("Thread %lx :\"%s\"\n",pthread_self(),str);
69   pthread_join(th1,NULL);
70   pthread_join(th2,NULL);
71   return 0;
72  }
```

说明：上面的程序是一个典型的使用静态变量来累加调用结果的库函数的例子，这种累加表现为将返回的字符串连接起来。对于单线程程序来说，我们可以定义累加函数如第9~14 行所示。在这个函数中定义了一个静态保存变量 accu。对于多线程来说，因为所有的线程都使用同一个地址来保存变量 accu，所以我们要使用线程特定数据的处理来为每一个线程分配不同的变量 accu。第 17 行定义了线程特定数据的关键字。第 18~20 行定义了静态变量，保证这个变量仅仅被分配一次空间。第 21~36 行实现了 str_accumulate 函数的线程安全版本。pthread_once 函数确保关键字仅被分配一次(第 24 行)，pthread_getspecific 函数获得与关键字相关的线程特定数据(第 25 行)，如果 accu 的初始值为 NULL，则首先为缓冲区分配空间(第 26~33 行)。这样就可以像一般程序中使用 accu 一样在线程中使用了。str_alloc_key 函数的功能是为线程特定数据区分配关键字(第 38~42 行)。str_alloc_destroy_accu 函数的功能是线程退出后释放缓冲区(第 44~48 行)，只有当线程特定数据区为 NULL时才调用该函数。process 是线程函数，其中调用了 str_accumulate 函数实现累加功能(第50~58 行)。在 main 函数中，创建了 2 个线程分别实现各自的累加功能(第 65~66 行)。同时在主线程中也调用了 str_accumulate 函数(第 64、67 行)。程序的执行结果为：

```
$./ex7
Thread b7e636b0 : allocated key 0
Thread b7e636b0 : allocating buffer at 0x804a008
Thread b7e636b0 :"Result of initial thread"
Thread b7e62b90 : allocating buffer at 0x804a530
Thread b7e62b90: "Result of first thread"
Thread b7e62b90 : freeing buffer at 0x804a530
Thread b7661b90 : allocating buffer at 0x804a530
Thread b7661b90: "Result of second thread"
Thread b7661b90 : freeing buffer at 0x804a530
```

从执行结果可以看出，虽然 3 个线程都使用同一缓冲区，但输出结果为各自特定数据，并不相同。

7.2.6 线程属性

每个 POSIX 线程有一个相连的属性对象来表示特性。线程的属性对象能与多个线程相连，POSIX 具有创建、配置和删除属性对象的函数。线程可以分组，并将相同属性与组中所有成员相连。当属性对象的一个特性改变时，组中所有实体具有新的特性。线程属性对象的类型是 pthread_attr_t，pthread_attr_t 在文件/usr/include/bits/pthreadtypes.h 中被定义，下面给出了它的定义：

```
typedef struct
{
 int     detachstate;
```

```
    int             schedpolicy;
    struct sched_param      schedparam;
    int             inheritsched;
    int             scope;
    size_tguardsize;
    int             stackaddr_set;
    void  *stackaddr;
    size_tstacksize;
} pthread_attr_t;
```

在该结构中，datachstate 表示线程的拆卸状态，schedpolicy 表示线程的调度策略，schedparam 表示线程的调度参数，inheritsched 表示线程的继承性，scope 表示线程的作用域，stackaddr 表示线程堆栈的位置，stacksize 表示线程堆栈的大小。

1. 线程属性对象的初始化和销毁函数

在使用一个线程属性对象之前，必须对其进行初始化，pthread_attr_init 函数完成对线程属性对象初始化；在使用完一个线程属性对象后，必须对其进行销毁，pthread_attr_destroy 函数完成对线程属性对象的销毁。这 2 个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

函数 pthread_attr_init 和 pthread_attr_destroy 都只有 1 个参数，此参数为一指向线程属性对象的指针。

这 2 个函数在调用成功时返回 0，失败时返回-1。

2. 线程堆栈大小相关函数

函数 pthread_attr_setstacksize 和 pthread_attr_getstacksize 分别用来设置和得到线程堆栈的大小，这 2 个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是堆栈大小或指向堆栈大小的指针。

这两个函数在成功调用时返回 0. 失败时返回-1。

3. 线程堆栈地址函数

函数 pthread_attr_setstackaddr 和 pthread_attr_getstackaddr 分别用来设置和得到线程堆栈的位置，这 2 个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stack_addr);
int phread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是堆栈地址或指向堆栈地址的指针。

这两个函数在成功调用时返回 0，失败时返回-1。

4. 线程的拆卸状态函数

函数 pthread_attr_setdetachstate 和 pthread_attr_getdetachstate 分别用来设置和得到线程的拆卸状态，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int phread_attr_get detachstate(const pthread_attr_t *attr, int* detachstate);
```

这两个函数具有 2 个参数，第 1 个是指向属性对象的指针，第 2 个是拆卸状态或指向拆卸状态的指针。拆卸状态可能的值是 PTHREAD_CREATE_JOINABLE 或是 PTHREAD_CREATE_DETACHED，默认值是前者。

在可联合的状态中，另外一个线程可以通过 pthread_join 函数来同步线程的终止，而且可以恢复线程的终止代码，但是有一些线程的资源在线程退出后并不会释放，这样其他线程在创建时可以重新利用这些资源。

在脱离状态下，线程的资源在线程结束后立刻释放，而且不能用 pthread_join 函数来同步线程的终止。

这 2 个函数在成功调用时返回 0，失败时返回-1。

5. 线程的作用域函数

函数 pthread_attr_setscope 和 pthread_attr_getscope 分别用来设置和得到线程的作用域，这 2 个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int phread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

这两个函数具有 2 个参数，第 1 个是指向属性对象的指针，第 2 个是作用域或指向作用域的指针。作用域控制线程是否在进程内或在系统级上竞争资源，可能的值是 PTHREAD_SCOPE_PROCESS 或是 PTHREAD_SCOPE_SYSTEM。系统默认值为：PTHREAD_SCOPE_SYSTEM。

这 2 个函数在成功调用时返回 0，失败时返回-1。

6. 线程的继承调度函数

继承调度的意思是当新创建一个线程时，线程的调度策略和调度参数是由 schedpolicy

和 schedparam 属性指定还是由创建它的父线程那里继承。函数 pthread_attr_setinheritsched 和 pthread_attr_getinheritsched 分别用来设置和得到线程的继承调度，这 2 个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int phread_attr_getinheritsched (const pthread_attr_t *attr, int *inherit);
```

这两个函数具有 2 个参数，第 1 个是指向属性对象的指针，第 2 个是继承调度或指向继承调度的指针。继承调度可能的值是 PTHREAD_EXPLICIT_SCHED 或是 PTHREAD_INHERIT_SCHED，分别对应上面 2 种情况。系统的默认值为 PTHREAD_EXPLICIT_SCHED。

这 2 个函数在成功调用时返回 0，失败时返回-1。

7. 线程的调度策略函数

函数 pthread_attr_setschedpolicy 和 pthread_attr_getschedpolicy 分别用来设置和得到线程的调度策略，这 2 个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int * policy);
```

这两个函数具有 2 个参数，第 1 个是指向属性对象的指针，第 2 个是调度策略或指向调度策略的指针。调度策略可能的值是先进先出(SCHED_FIFO)、轮转法 (SCHED_RR)，或是其他未定义(SCHED_OTHER)。调度策略的默认值是 SCHED_OTHER。

调度策略 SCHED_RR 和 SCHED_FIFO 仅仅对有超级用户权限的进程才有效。

这 2 个函数在成功调用时返回 0，失败时返回-1。

8. 线程的调度参数函数

函数 pthread_attr_setschedparam 和 pthread_attr_getschedparam 分别用来设置和得到线程的调度参数，这 2 个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setschedparam (pthread_attr_t *attr,const struct sched_param *param);
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, struct sched_param *param);
```

这两个函数具有 2 个参数，第 1 个是指向属性对象的指针，第 2 个参数是 sched_param 结构或指向该结构的指针。结构 sched_param 在文件/usr/include/bits/sched.h 中定义，如下所示：

```
struct sched_param
{
 int sched_priority;
};
```

结构 sched_param 的子成员 sched_priority 控制一个优先权值，大的优先权值对应高的优先权。系统默认的调度参数是：优先级 0。

如果线程的调度策略是 SCHED_OTHER，那么这个参数就可以忽略。只有当线程的调度策略是 SCHED_RR 或者 SCHED_FIFO 时，这个参数才有用。

这 2 个函数在成功调用时返回 0，失败时返回-1。

以上介绍了与线程属性有关的函数，下面来看一个例子。

例 7-8 线程属性相关函数示例。

```
1   /*ex8.c*/
2   #include <stdio.h>
3   #include <errno.h>
4   #include <pthread.h>
5   #include <unistd.h>
6
7   void *my_thread(void *arg)
8   {
9     int retval=0;
10    pthread_attr_t attr;
11    struct sched_param param;
12    size_tstacksize;
13    int         detachstate;
14    int         scope;
15    int         inherit;
16    int         policy;
17    if(pthread_attr_init(&attr)==0)
18    {
19         if(pthread_attr_getstacksize(&attr,&stacksize)==0)
20         {
21              printf("StackSize: %d\n",stacksize);
22         }
23         if(pthread_attr_getdetachstate(&attr, &detachstate)==0)
24         {
25              if(detachstate==PTHREAD_CREATE_JOINABLE)
26                   printf("DetachState :PTHREAD_CREATE_JOINABLE\n");
27              if(detachstate==PTHREAD_CREATE_DETACHED)
28                   printf("DetachState :PTHREAD_CREATE_DETACHED\n");
29         }
30         if(pthread_attr_getscope(&attr, &scope)==0)
31         {
32              if(scope==PTHREAD_SCOPE_PROCESS)
33                   printf("Scope :PTHREAD_SCOPE_PROCESS\n");
34              if(detachstate==PTHREAD_SCOPE_SYSTEM)
```

```
35                  printf("Scopee :PTHREAD_SCOPE_SYSTEM\n");
36          }
37          if(pthread_attr_getinheritsched(&attr, &inherit)==0)
38          {
39              if(inherit==PTHREAD_INHERIT_SCHED)
40                  printf("InheritSched:PTHREAD_INHERIT_SCHED\n");
41              if(inherit==PTHREAD_EXPLICIT_SCHED)
42                  printf("InheritSched:PTHREAD_EXPLICIT_SCHED\n");
43          }
44          if(pthread_attr_getschedpolicy(&attr, &policy)==0)
45          {
46              if(policy==SCHED_FIFO)
47                  printf("SchedPolicy:SCHED_FIFO\n");
48              if(policy==SCHED_RR)
49                  printf("SchedPolicy:SCHED_RR\n");
50              else
51                  printf("SchedPolicy:SCHED_OTHER\n");
52          }
53          if(pthread_attr_getschedparam(&attr, &param)==0)
54          {
55              printf("SchedPriority:%d\n",param.sched_priority);
56          }
57          pthread_attr_destroy(&attr);
58      }
59      pthread_exit(&retval);
60  }
61
62  int main()
63  {
64      int         count;
65      pthread_t   thread;
66      int         *retval;
67      if(pthread_create(&thread,NULL,my_thread,(void *)NULL)!=0)
68      {
69          printf("Count not create thread! \n");
70          return -1;
71      }
72      if(pthread_join(thread,(void **)(&retval))!=0)
73      {
74          printf("No thread to join! \n");
75          return -2;
76      }
77      return 0;
```

```
78 }
```

说明：上面程序的功能是调用 pthread_create 创建一个线程(第 67 行)，然后在创建的线程中，分别调用线程属性相关函数得到线程的各个属性，并将它们打印输出(第 7~60 行)。

程序执行结果如下所示：

```
$ ./ex8
StackSize: 8388608
DetachState :PTHREAD_CREATE_JOINABLE
Scopee :PTHREAD_SCOPE_SYSTEM
InheritSched:PTHREAD_INHERIT_SCHED
SchedPolicy:SCHED_OTHER
SchedPriority:0
```

从程序执行结果可以看出新创建的线程属性都是采用系统默认值。

7.3 小 结

本章介绍了 Linux 中线程以及线程函数的使用方法。使用线程一个明显的好处是提高应用程序的响应速度，而且线程之间通信很方便。读者需要掌握线程的一些概念，包括线程与进程之间的区别，线程模型的优点，线程的创建与结束，线程之间的同步、取消线程和取消处理程序、线程特定数据的处理以及线程属性相关操作等。完整地理解 Linux 线程操作是非常重要的，读者需要对以上系统函数熟练掌握。

习 题

一、填空题

1. 线程可分为______态线程和______态线程。
2. 如果线程可在进程执行期间的任意时刻被创建，并且线程的数量事先没有必要指定，这样的线程称为______线程。
3. 按照 POSIX 标准，POSIX 提供了两种类型的同步机制，它们是______和______。
4. 互斥锁的特点是______、______、______。
5. 每个 POSIX 线程由一个相连的______来表示特性。

二、选择题

1. 在 POSIX 中，线程是用______动态地创建的。

   (A) pthread_self　(B) pthread_create　(C) create_pthread　(D) pthread_new

2. 要结束一个线程，需要调用函数______。

   (A) exit　(B) pthread_quit　(C) pthread_exit　(D) return

3. 可以使用下面的函数______将一个线程挂起。

   (A)pthread_self　(B) pthread_pause　(C) pthread_join　(D) pthread_exit

4. ______用来初始化一个互斥锁。

   (A) pthread_mutex_init　(B)pthtead_mutex_create

   (C) pthtead_mutex_begin　(D) pthtead_mutex_lock

5. 在使用一个线程属性对象之前，必须对其进行初始化，______函数完成对线程属性对象初始化。

   (A) pthread_init (B) pthread_attr_create (C) pthread_attr_destroy (D) pthread_attr_init

三、上机题

1. 编写一个包含 2 个线程的程序，在主线程中创建一个全局变量并初始化为 0，在另一个线程对这个全局变量进行递加运算，并在结束时向主线程返回一个结果，由主线程打印输出。

2. 编写一个包含 2 个线程的程序，在主线程中接受键盘输入，并把输入字符放入缓冲区中，在缓冲区满后，由另一个线程输出缓冲区的内容，用互斥锁实现二者之间的同步。

3. 用条件变量重新编写例 7-2 程序。

4. 编写一个程序，在主线程中创建一个新线程，在主线程中得到新线程的各个属性，并在主线程中将它们打印输出。



网 络 编 程

Linux 本身就是一个网络的产物——它的第一个版本就放在网上供人们自由下载，电脑爱好者和黑客们通过网络对它进行修改、完善，更多的用户通过网络了解并免费使用它。没有网络，就不可能有 Linux 如此辉煌的今天。反过来，Linux 也对网络提供了强大的支持。强大的通信和联网功能一直是 Linux 被人们津津乐道的特点；Linux 的网络连接在内核中完成，因此十分稳定；能支持多种网络协议，常用的有 TCP/IP、IPX、DDP 等，还有最新的 IPv6 等；Shell 提供功能强大的联网命令，例如 ftp、telnet 等。因此，使用 Linux，就不能不涉及网络；而学习 Linux 环境下的编程，也不能不学习网络编程。

本章简单介绍 Linux 下的网络编程基本知识。只用一章的篇幅是不可能把 Linux 丰富的网络功能都讨论完的，而这一章的目的只是把网络方面主要的程序接口介绍给读者。掌握了这些程序设计接口，就能开始编写网络程序了。将要学习的内容主要包括套接字连接的操作原理、套接字的属性、地址和通信、客户和服务器等。

8.1 概　　述

计算机网络的研制开始于 20 世纪 60 年代中期，至今已有 40 多年的历史。网络技术发展及应用已经十分普及，并渗透到了各个领域，正在日益显示它给信息化社会带来的影响和深远意义。

为了减少网络在设计上的复杂性，大多数网络都分成若干层。每一层都向其上层提供一定的服务，这些服务的具体实现对上层是不可见的。相邻两层之间使用一定的原语进行交互。这样的设计提供了层在具体实现上的独立性。同样的层在不同的计算机或是不同的操作系统上可能有不同的实现，但只要它正确实现与上层及下层的交互界面的原语，提供与以前一样的职务，就可以保证网络通信的正常进行。

网络通信是在两个通信实体之间进行的。这两个实体相应的层之间使用协议来互相交换信息。协议是同层之间约定的进行交流的规则。举例来说，两个不同国家的学者通过电话进行学术问题交流，他们一个说希伯莱语，一个说汉语，各自的翻译除了会本国语言外都会英语和法语。他们怎样进行交流呢?显然可以使用下列过程：说希伯莱语的学者先用希伯莱语告诉他的翻译要说的内容，然后他的翻译将其译成英语，再通过电话告诉另外一个学者的翻译。这个翻译将英语再译成汉语，然后告诉说汉语的学者。这个过程可以看作两个学者通过一个“翻译网络”进行通信的过程。翻译们之间具体使用什么语言进行交流，就是翻译层(如果把翻译者作网络中的一个层的话)的协议。在这个例子中，翻译之间使用的协议是英语，但是如果翻译们约定为用法语(另一协议)，并不会影响两个学者之间的交流。这也就是前面所说的层在具体实现上的独立性。类似地，可以把电话也看成是网络中的一层，它们之间的协议就是语音信号，而电话之间是用什么样的调制方式的电波通信不会影响到它们传递的语音信号。层与协议一起构成了网络体系结构。

在网络化技术迅速发展的今天， TCP/IP 协议立下了汗马功劳。TCP/IP(传输控制协议/网际协议 Transmission Control Protocol/Internet Protocol)，实际上是一个由多种协议组成的协议簇，它定义了计算机通过网络互相通信及协议簇各层次之间通信的规范。

TCP/IP 最初是在由美国政府资助的美国高等研究计划署的网络 ARPANET 上发展起来的，该网络用于支持美国军事和计算机科学研究，正是由它提出了报文交换和网络分层概念。1988 年以后，ARPANET 由其继任者——美国国家科学基金会的 NSFNET 所取代，而 NSFNET 和全世界数以万计的局域网和城域网共同连接成了一个巨大的联合体——因特网(Internet)，举世闻名的万维网(World Wide Web)也是来自于 ARPAnet 并完全采用 TCP/IP 协议簇。UNIX 被广泛应用于 ARPANET，它的第一个网络版是 4.3 BSD(Berkeley Software Distribution)， 该版本支持 BSD 的套接字和全部的 TCP/IP 协议，Linux 的网络功能即是基于这个版本实现的。Linux 之所以以该 4.3 BSD 版本为模型，是因为这个版本广为流行，并且它支持 Linux 与其他 UNIX 平台之间应用程序的移植。

8.2 TCP/IP 基 础

TCP/IP 协议是一组在网络中提供可靠数据传输和无连接数据服务的协议。其中提供可靠数据传输的协议称为传输控制协议 TCP，而提供无连接数据包服务的协议叫做网际协议 IP。但是 TCP/IP 协议并不是只有 TCP 和 IP 两个协议，而是包含很多其他协议的一个网络协议的集合。

使用 TCP/IP 协议的网络提供的主要服务有电子邮件、文件传送、远程登录、网络文件系统、电视会议系统和万维网等。

8.2.1　参考模型

TCP/IP 协议参考模型共分四层，如图 8-1 所示。

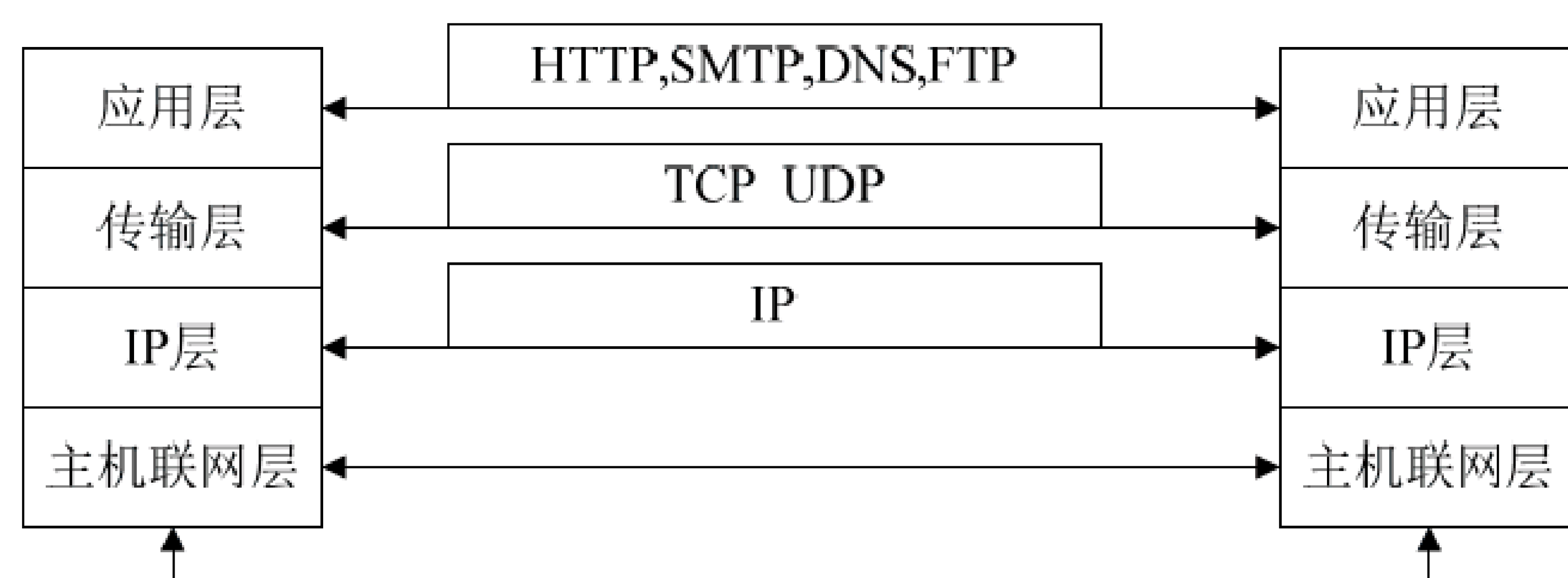


图 8-1　TCP/IP 参考模型及其协议族

(1) 应用层

应用层提供各种常用的高层协议：FTP(文件传输)、TLENET(远程登录)、SMTP(简单邮件传送)、HTTP(超文本传输)、DNS(域名服务)等。

(2) 传输层

提供两种端到端的会话协议。端到端是指传输层协议是从自己的一个端口发送数据到对方的一个端口。端口是为了方便同一主机上同时执行多个网络程序而设置的。因此在传输层的通信中，除了要指明对方的 IP 地址，还要指定对方应用程序所使用的端口。以科苑星空 BBS 所在的主机 210.77.16.7 为例，当连接端口 23 时，将进入远程登录使用方式；而当连接端口 80 时，将进入 www 使用方式。大多数的因特网服务器都不止提供一项服务，根据端口的不同就能区分同一主机上的不同服务。

TCP 是面向连接的协议，提供无差错的字节流的可靠传递。要发送的字节流被分成若干块顺序传递到 IP 层，到达目的地后再由对方的 TCP 顺序组装起来。如果数据报在这一过程中丢失，发送方 TCP 层还将负责重发。TCP 还进行流量控制，防止发送速度大于接收速度而发生数据报丢失的现象。

UDP(User Datagram Protocol)不是可靠地面向连接的协议，它仅将数据打包送出，不保证数据报一定到达接收方，接收方 UDP 也不会调整数据报的顺序。这一协议主要用于强调数据传递的速度而不是可靠性的场合，如语音和视频信息的传送。

(3) 网际协议(Internet Protocol)层

IP 这个名词可能是最广为人知的网络名词了。现在因特网上任何一台主机都有一个独一无二的 IP 地址。目前的 IP 地址是由 4 个字节组成的，如 210.77.16.7，转换为二进制为 11010010. 01001101.00010000.00000111。IP 地址格式的规定就包含在 IP 协议中。

IP 协议定义了 4 种主要的地址类：A、B、C 和 D 类。

- A 类地址：第一位固定为 0，第一个字节(前 8 位)为网络标识符，用来标识网络，其余 3 个字节用来标识网络中的主机。因此最多有 127 个 A 类网络，每个 A 类网络可以容纳 1700 万台主机。

- B 类地址：前两位固定为 10，第一个和第二个字节(前 16 位)为网络标识符，用来标识网络，其余 2 个字节用来标识网络中的主机。因此最多有 16000 个 B 类网络，每个 B 类网络可以容纳 65000 台主机。
- C 类地址：前三位固定为 110，前 3 个字节(前 24 位)为网络标识符，用来标识网络，最后一个字节用来标识网络中的主机。因此最多有 200 万个 C 类网络，每个 C 类网络可以容纳 254 台主机。
- D 类地址：前 4 位固定为 1110。D 类地址是多目地址，标识在网络上运行分布式应用的一群主机。因此，D 类主机并不标识一个在线的主机。

对于一个给定的 IP 地址，我们可以判定它属于哪类地址、网络地址和节点地址。

例如，地址 166.111.111.5 的首字节在 128~191 之间，因此该地址为 B 类地址，网络地址为 166.111，主机地址为 111.5。

IP 地址由一些数字组成，比较难记住，而记住一个名字则相对容易多了，因此，为了方便使用，必须找到某种机制将网络名称转换成 IP 地址。在 Linux 中，这些名称在文件/etc/hosts 中记录，或者可以要求 DNS(域名服务器)来对名称进行解析。如果由 DNS 来解析地址，那么当地主机必须知道一个或多个 DNS 的 IP 地址，这些 DNS 在文件/etc/resolv.conf 中记录。

IP 层是 TCP/IP 模型中最关键的一环。这一层上实现了 IP 分组(Packet)在网络上的点到点传送，也就是说，发送 IP 分组只需指明它要到达的地址，而不必关心它经过什么路径到达。一般说来，如果不是与所连接的对象处于同一局域网内，很难知道发送的 IP 分组将如何到达对方。IP 分组在因特网上传输所经过的路径叫路由。连接因特网各子网的服务器，将负责把 IP 分组从一个子网发送到另一个子网，这样的服务器称为路由器。路由的选择将在 IP 分组到达各路由器时进行，路由器根据它所储存的网络信息和 IP 分组所携带的目的地址信息，决定向何处发送 IP 分组(如果路由器与目的地址直接连接，则将直接发送到该机器；如果没有直接连接，则根据一定算法选择发送到与本路由器相连的其他中转路由器)，这一分组转发过程与发送方和接收方无关。这样就实现了异种网路之间的分组交换，使得整个因特网看起来像一个大的单一的网络。当前最广为使用的 IP 协议是 IPV4 协议，定义的是 32 位的 IP 地址，但随着 Internet 的迅速发展，32 位的地址已经不够用了，为此又提出了使用 128 位地址的 IPv6 协议。

(4) 主机联网层

这一层在 TCP/IP 模型中没有详细说明。这一层的实现与具体的网络有关，没有具体的协议。只要在这一层的软硬件能正确地接收发送 IP 分组就足够了。

8.2.2 Linux 中 TCP/IP 网络的层结构

图 8-2 显示了 Linux 系统网络实现的分层结构。最上层是网络应用程序层，如 WWW、FTP、E-mail 等网络应用程序位于这一层。网络应用程序层通过 BSD 套接字进行数据传输。BSD 套接字是最早的网络通信的实现，它由一个只处理 BSD 套接字的管理软件支持。其

下面是 INET 套接字层，它管理 TCP 协议和 UDP 协议的通信末端。UDP(User Datagram Protocol)是无连接的协议，而 TCP 则是一个可靠的端到端协议。当网络中传送一个 UDP 数据包时，Linux 系统不知道也不关心这些 UDP 数据包是否安全地到达目的节点。TCP 数据包是编号的，同时 TCP 传输的两端都要确认数据包的正确性。IP 协议层是用来实现网间协议的，其中的代码要为上一层数据准备 IP 数据头，并且要决定如何把接收到的 IP 数据包传送到 TCP 协议层或者 UDP 协议层。在 IP 协议层的下方是支持整个 Linux 网络系统的网络设备，如 PPP、以太网(Ethernet)等。网络设备并不完全等同于物理设备，因为一些网络设备，如回馈设备是完全由软件实现的。和其他使用 mknod 命令创建的 Linux 系统的标准设备不同，网络设备只有在软件检测到和初始化这些设备时才在系统中出现。当构建系统内核时，即使系统中有相应的以太网设备驱动程序，也只能看到/dev/eth0。

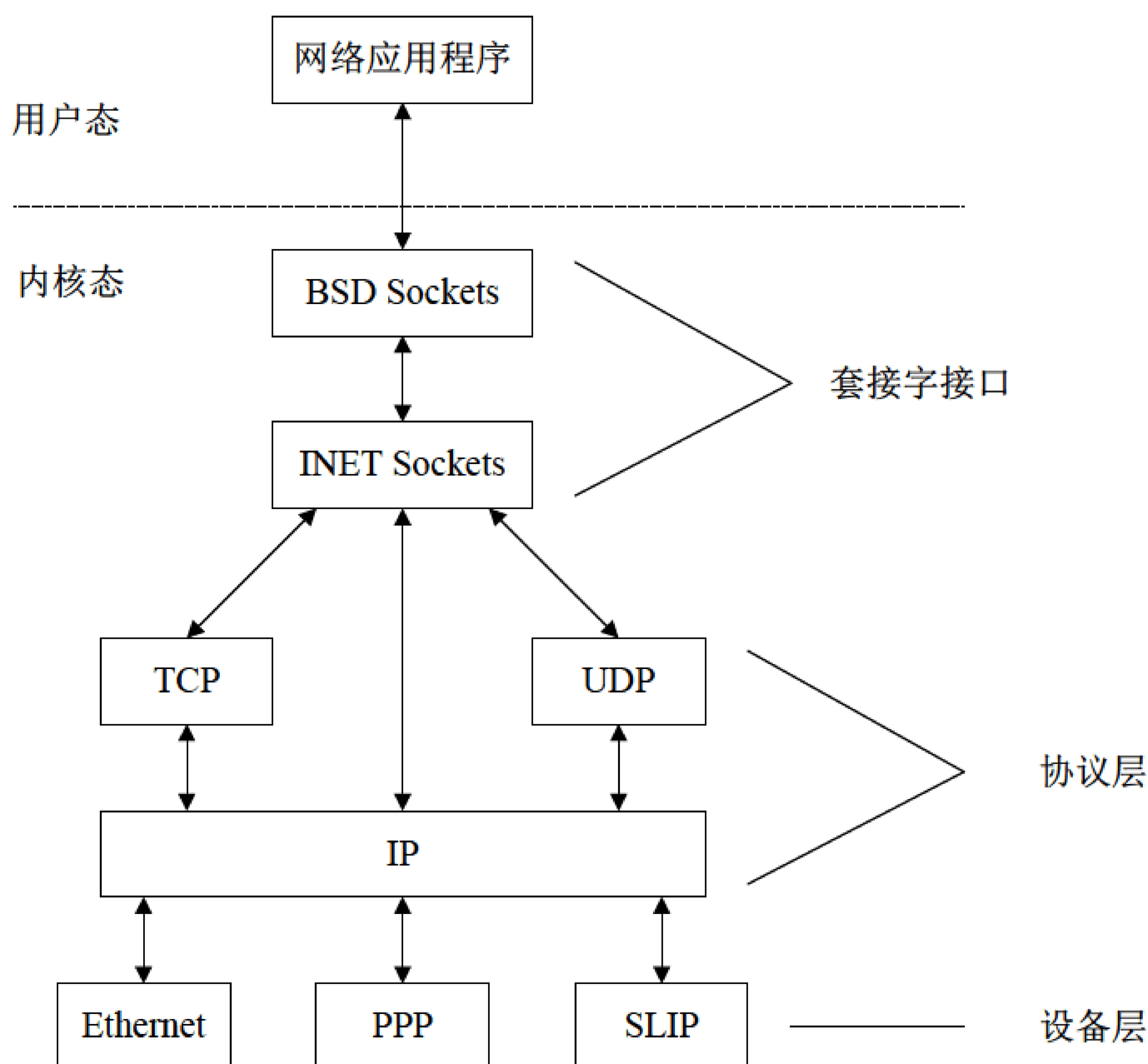


图 8-2 Linux 网络层分布结构

8.3 BSD 套接字接口

BSD 套接字接口是 BSD 的进程间通信方式，它不仅支持各种形式的网络应用，而且它还是一种进程间通信的机制。一个套接字描述一个通信连接的一端，两个相互通信的进程，每个都需要一个套接字描述它们之间的通信连接的端点。套接字可以看成是一种特殊的管道，与管道不同的是套接字所能容纳的数据不受限制。

Linux BSD 支持如下类型的套接字，它们是：

(1) Stream(数据流)。这个套接字提供了两个方向的序列数据流，这些数据流保证在传输过程中数据不丢失、破坏或重复。数据流套接字由Internet(INET)地址簇的TCP协议所支持。

(2) Datagram(数据报)。这个套接字也提供两个方向上的数据传送，但不像数据流套接字，它们不提供消息到达的保证。即使到达也不保证这些数据报按照一定的顺序到达或丢失、重复。这种类型的套接字由Internet地址簇的UDP协议所支持。

(3) Raw(原始套接字)。这种类型的套接字允许进程直接访问底层协议。例如，可以为以太网设备打开一个Raw Socket，以使用原始IP数据进行传输。

(4) Reliable Delivered Message(可靠传递消息)。它非常像数据报套接字，但是保证数据的可靠传输。

(5) Sequenced Packets(顺序数据报)。它像数据流套接字，但数据包的大小固定。

(6) Packet(包)。它不是标准的BSD套接字类型，它是一个Linux特定的扩展，允许进程在设备层直接访问Packet。

利用套接字进行通信的进程采用客户机/服务器(C/S)模式。服务器提供服务而客户机则使用服务器提供的服务。使用套接字的服务器首先建立一个套接字，然后用一个名称对这个套接字进行绑定。这个名称的格式独立于套接字的地址簇，它是有效的服务器的本地地址。套接字的名称或地址由sockaddr结构来指定，一个INET套接字由一个IP端口地址与之绑定。常用服务的注册端口可以在/etc/services中看到，例如端口80是Web服务器的特定端口。当给一个套接字绑定一个地址后，服务器侦听输入请求指定的绑定地址的连接。客户建立一个套接字和一个基于它的连接请求，这个连接请求指定目的服务器的地址。对一个INET套接字来讲，服务器的地址是它的IP地址和端口号。这些传入的请求必须通过各种不同的协议层向上找到自己的通路，然后等待服务器侦听套接字。一旦服务器收到请求，它要么接收要么拒绝。如果传入请求被接收，服务器必须建立一个新的套接字用来接收。如果一个套接字已经用来侦听传入的连接请求，那么它不能用来支持一个连接。

8.4　客户机/服务器(C/S)模式

TCP/IP允许程序员在两个应用程序之间建立通信并来回传递数据，提供一种对等通信，这种对等应用程序可以在同一台机器上，也可以在不同的机器上运行。尽管TCP/IP指明了数据是如何在一对正在通信的应用程序间传递的，但是它并没有规定对等的应用程序在什么时间进行交互以及为什么要进行交互，也没有规定程序员在一个分布式环境下应该如何组织这些应用程序。实践中，有一种有组织的方法在使用的TCP/IP中占据着主要地位，那就是客户机/服务器(C/S)模式，现在网络上的绝大多数通信应用程序都使用这种机制。

客户机/服务器模式要求每个应用程序由两个部分组成：一个部分负责启动通信，另一个部分负责对它进行应答。它们通常运行在不同的主机上，分别称为客户机和服务器。服

务器是指在网络上可提供服务的任何程序；客户机是指用户为了得到某种服务所需要运行的应用程序。一个服务器接收网络上客户机的请求，完成服务后将结果返回给客户机，它们之间的关系如图 8-3 所示。



图 8-3　客户机/服务器关系图

服务器能完成简单和复杂的任务，一台主机可以同时运行多个服务器程序，一个服务器程序可以同时接受一个或是多个客户的请求，当客户发送某个服务请求时，服务器将其在提供该服务的端口排队，然后从队列中提取请求，为每个请求创建一个子进程，由子进程来处理具体的服务细节。

通常情况下，服务器包括两个部分：主程序和从程序。主程序负责接收来自客户的请求，从程序一般有几个，它们负责处理各个客户请求。

主、从程序工作过程如图 8-4 所示。

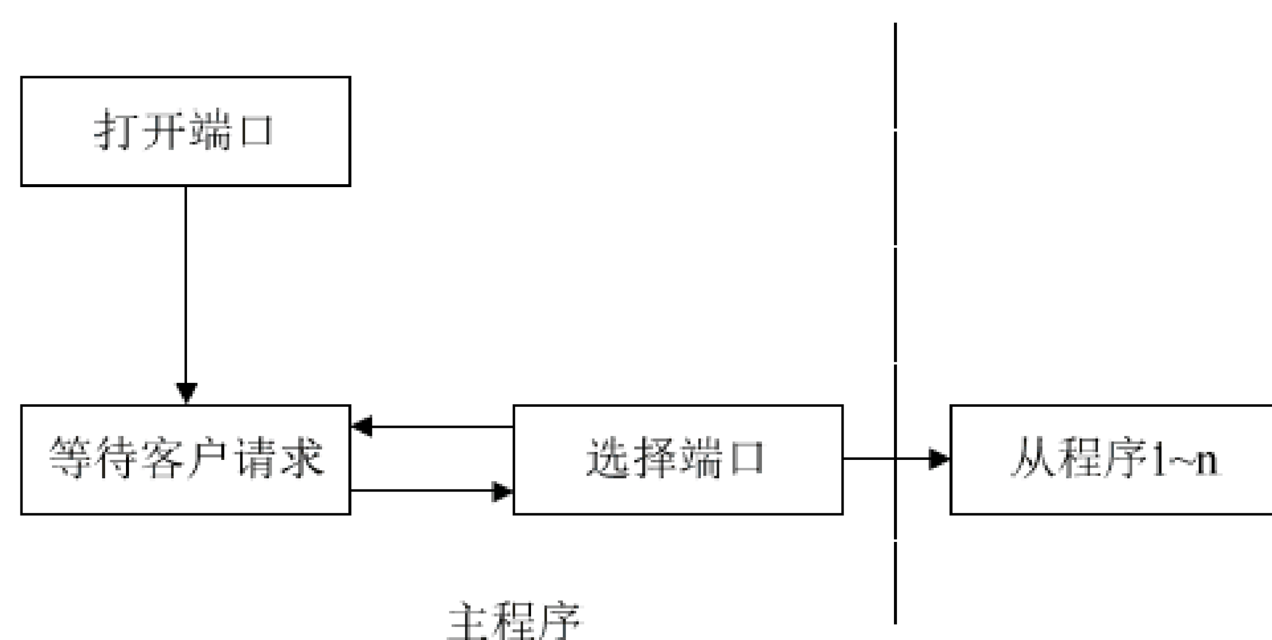


图 8-4　服务器工作过程图

如果客户请求所指的端口不是已知的端口，则应为它请求分配一个临时端口，然后启动从程序，等待新的客户请求。从程序通常是个子进程，处理完一个客户请求后就中止并返回结果。

服务器通常是作为应用程序，而不是主机。服务器作为应用程序的优点是：它们可以在任何一个支持该通信协议的计算机系统上运行，这样不同的服务器可以同在一个分时系统上运行，或在一台个人计算机上运行，网络编程人员也可以在同一台机器上同时运行客户机和服务器，为测试和调试软件带来方便。如果一台计算机的主要任务是支持某个服务器程序，那么服务器这一名称不但是指服务器程序，也指计算机。同理，客户机也是一样。

8.5　套接字网络编程

前面介绍了网络相关的一些基本内容，本节介绍 Linux 下的套接字网络编程。

8.5.1 套接字编程的基本流程

前面已经说过，Linux 支持 6 种类型的套接字接口，其中最常用的是 2 种：数据流套接字和数据报套接字。

数据流套接字定义了一种可靠的面向连接的服务，实现了无差错无重复的顺序数据传输。数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠、无差错。

无连接服务器一般都是面向事务处理的，一个请求、一个应答就完成了客户程序与服务程序之间的相互作用。若使用无连接的套接字编程，程序的流程可以用图 8-5 表示。

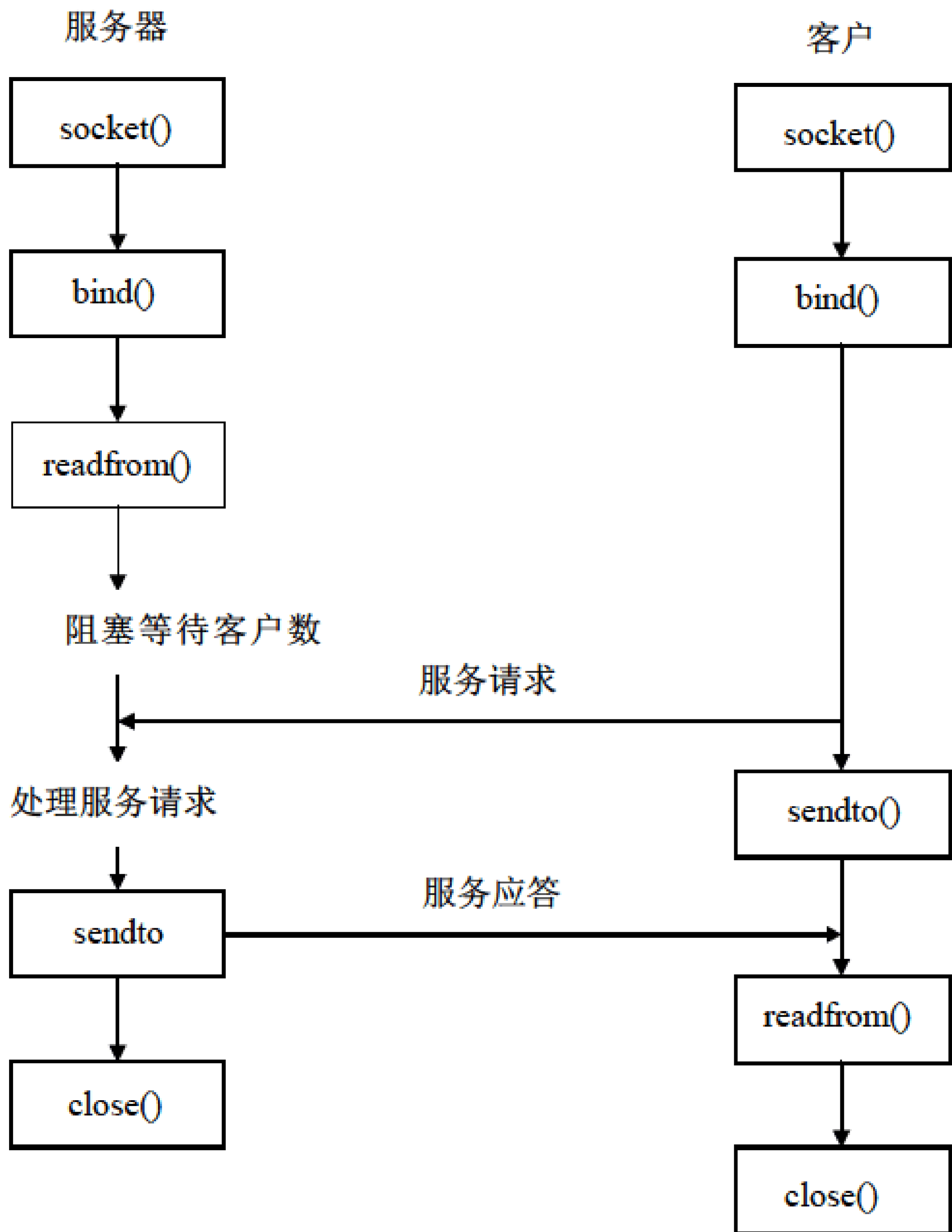


图 8-5　无连接套接字应用程序流程图

面向连接服务器处理的请求往往比较复杂，不是一来一去的请求应答所能解决的，往往是并发服务器。

面向连接的套接字工作过程如下：服务器首先启动，通过调用 socket 函数建立一个套接字，然后调用 bind 将该套接字和本地网络地址联系在一起，再调用 listen 使套接字做好侦听的准备，并规定它的请求队列的长度，之后就调用 accept 来接收连接。客户在建立套接字后就可调用 connect 和服务器建立连接。连接一旦建立，客户机和服务器之间就可以通过调用 read 和 write 来发送和接收数据。最后，待数据传送结束后，双方调用 close 关闭套接字。

图 8-6 是面向连接的套接字应用程序的流程图。

Linux 支持伯克利(BSD)风格的套接字编程，同时支持面向连接和无连接类型的套接

字。在面向连接的通信中服务器和客户在交换数据之前先要建立一个连接。在无连接的通信中数据作为消息的一部分被交换。无论哪一种方式，服务器总是最先启动，把自己绑定在一个套接字上，然后去侦听消息。服务器究竟怎样试图去侦听消息，取决于编程所设定的连接类型。

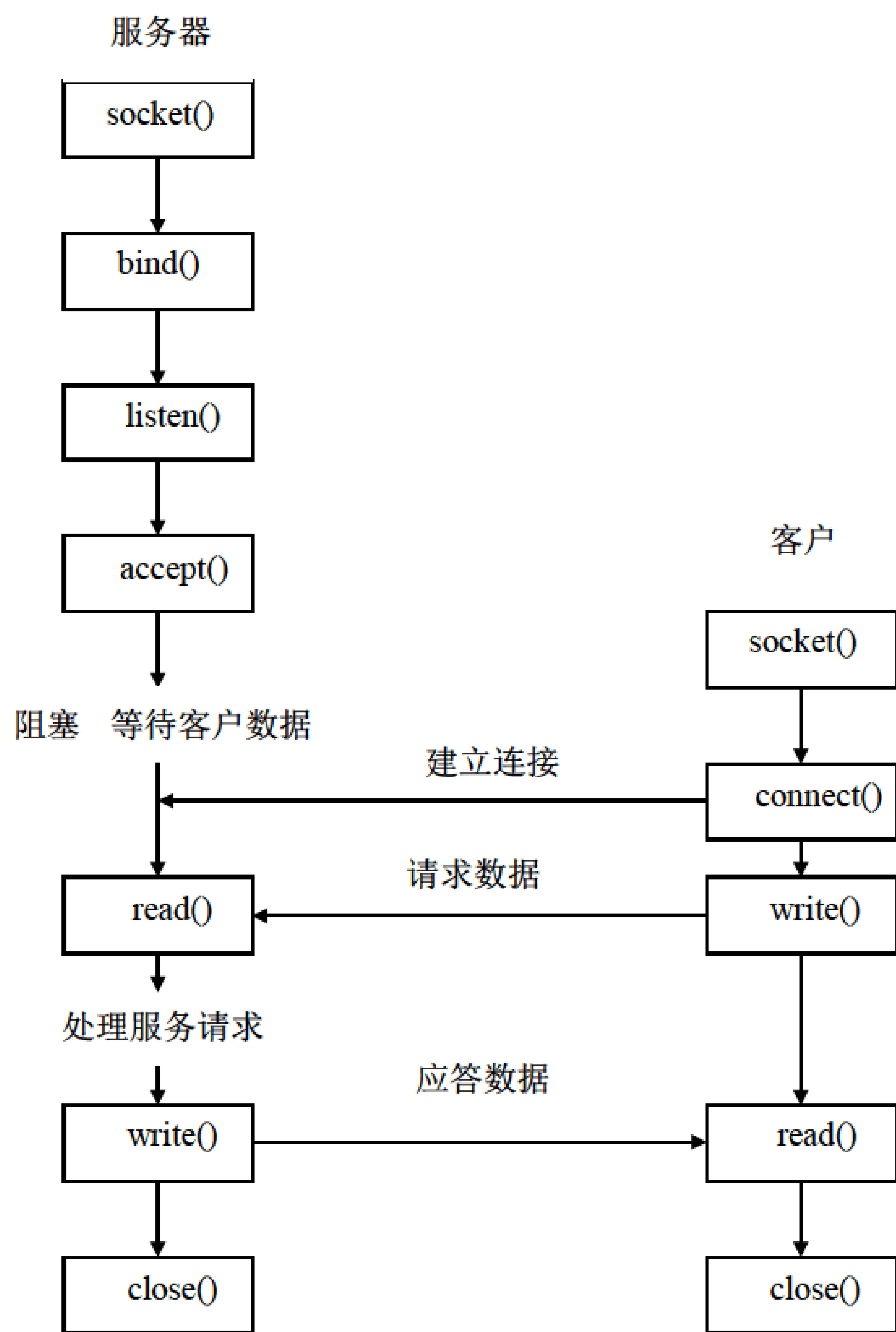


图 8-6　面向连接套接字应用程序流程图

8.5.2　套接字地址

Linux 系统的套接字是一个通用的网络编程接口，它支持多种协议，每一种协议使用不同的套接字地址结构。为了保持套接字函数调用参数的一致性，Linux 系统定义了一种通用的套接字地址结构，在系统头文件<sys/socket.h>中定义如下：

```
struct osockaddr
  {
    unsigned short int sa_family;              /*地址类型，AF_XXX */
    unsigned char sa_data[14];                  /*14 字节的协议地址  */
  };
```

其中 sa_family 为套接字的协议簇地址类型，如 TCP/IP 协议簇的地址类型为 AF_INET；sa_data 中存储具体的协议地址，不同的协议簇有不同的地址格式。

套接字支持的每种协议簇都定义了自己的套接字地址结构，以前缀 sockaddr 开始，TCP/IP 协议的套接字地址结构是 sockaddr_in ，在系统头文件<netinet/in.h>中定义：

```
struct in_addr
  {
    in_addr_t s_addr;
  };
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;                /* 端口号 */
    struct in_addr sin_addr;           /* Internet 地址   */

    /* 填充部分，未用 */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                   __SOCKADDR_COMMON_SIZE -
                   sizeof (in_port_t) -
                   sizeof (struct in_addr)];
  };
```

使用这个地址结构要注意：

(1) 结构 sockaddr_in 中的 TCP 或 UDP 端口号 sin_port 和 IP 地址 sin_addr 都是以网络字节顺序存储的。下一小节中将讨论如何将主机字节顺序的数据转换成网络字节顺序。

(2) 32 位的 IP 地址可以用两种不同的方法引用。例如，假设定义变量 servaddr 为 Internet 套接字地址结构，那么可以用 servaddr.sin_addr 或 servaddr.sin_addr.s_addr 来引用这个 IP 地址，需要注意的是，前一种引用是结构类型(struct in_addr)的数据，而后一种引用是整数类型的数据；当将 IP 地址作为函数参数使用时，需要明确使用哪种类型的数据，因为编译器对结构类型参数和整数类型参数的处理方式不一样。

(3) sin_zero 成员未被使用，它是为了和通用套接字地址(struct sockaddr)保持一致而引入的。在编程时，一般将它设置为 0。通常的做法是在填充结构 sockaddr_in 的内容之前将整个结构变量清零。

(4) 套接字地址结构仅供本机 TCP 协议记录套接字信息而用，这个结构变量本身是不在网络上传输的。但是它的某些内容，如 IP 地址和端口号是在网络上传输的，这也是这两部分数据需要转换成网络字节顺序的原因。

在设置结构 sockaddr_in 中的 IP 地址时，需要将字符串形式表示的 IP 地址转换成二进制形式。以下函数可以处理这个问题：

```
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in arrd *inp);
unsigned long int inet_addr(const char *cp);
char * inet_ntoa(struct in_addr in);
```

这 3 个函数将数字点形式表示的字符中 IP 地址与 32 位网络字节顺序的二进制形式的 IP 地址进行转换，如数字点形式表示的 IP 地址 192.168.0.10，对应于二进制形式表示的 IP 地址 C0A8000A。函数 inet_aton 将字符形式的 IP 地址转换成二进制形式的 IP 地址，成功时返回 1，否则返回 0，转换后的 IP 地址存储在参数 inp 中，函数 inet_addr 与函数 inet_aton 功能相同，但是转换结果在返回值中返回，错误时返回常量 INADDR_NONE。函数 inet_addr 已经过时，编程时应该使用函数 inet_aton，这是因为函数 inet_addr 不能处理广播地址 255.255.255.255。因为函数 inet_addr 错误时返回的值 INADDR_NONE 和广播地址 255.255.255.255 相同，32 位都是 l(即整数－1)。函数 inet_ntoa 将 32 位二进制形式的 IP 地址转换为数字点形式的 IP 地址，结果在函数返回值中返回。

8.5.3　字节顺序

网络中存在多种类型的机器，如基于 Intel 芯片的 PC 机和基于 RISC 芯片的工作站。这些不同类型的机器表示数据的字节顺序是不同的。考虑一个 16 位的整数 Al03，它由 2 个字节组成，高位子节是 A1，低位字节是 03。在内存中可以有两种方式来存储这个整数：低位字节存储在这个整数的开始地址位置(如图 8-7(a)所示)，或者高位字节存储在开始地址位置(如图 8-7(b)所示)。第一种字节顺序是 little-endian 方式，基于 Intel 芯片的机器采用的是这种方式；第二种字节顺序是 big-endian 方式，大多数基于 RISC 芯片的机器采用的是这种方式。主机存储数据的顺序称为主机字节顺序。

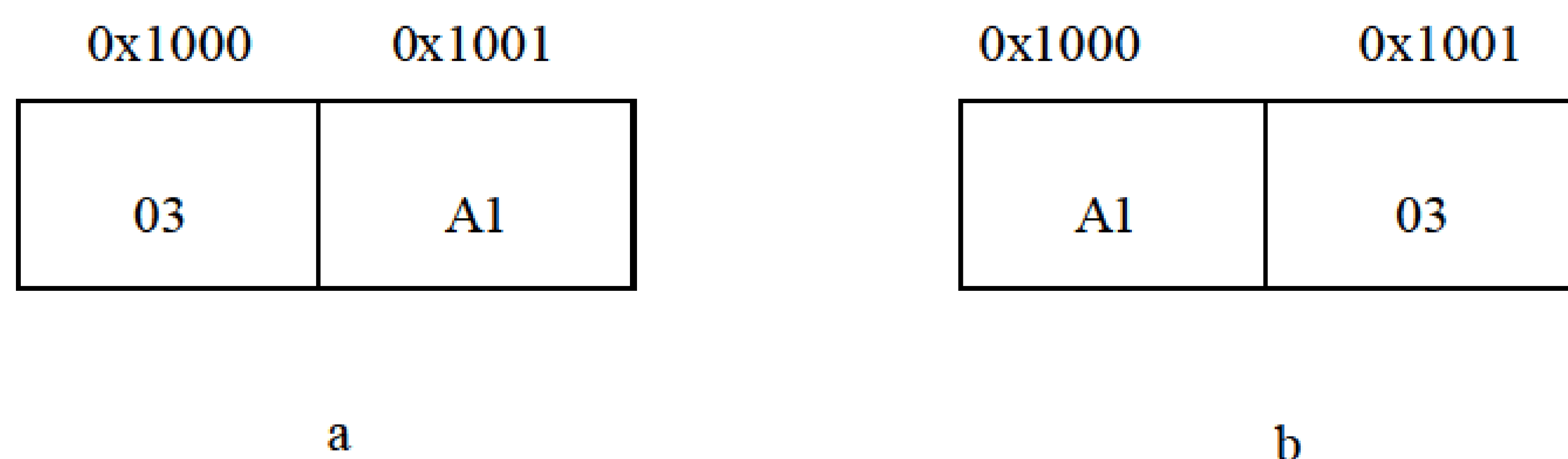


图 8-7　主机字节顺序

可以用下面的程序来检测系统采用的是什么字节顺序。

例 8-1　检测系统字节顺序。

```
1    /* ex1.c */
2    #include <sys/utsname.h>
3    #include <unistd.h>
```

```
4     #include <stdio.h>
5
6     int main()
7     {
8             union
9             {
10                    short inum;
11                    char c[sizeof(short)];
12            } un;
13            struct utsname     uts;
14            un.inum=0x0102;
15            if(uname(&uts)<0)
16            {
17                    printf("Could not get host information .\n");
18                    return -1;
19            }
20            printf("%s -%s-%s:\n",uts.machine, uts.sysname, uts.release);
21            if(sizeof(short)!=2)
22            {
23                    printf("sizeof short =%d\n", sizeof(short));
24                    return 0;
25            }
26            if(un.c[0]==1 && un.c[1]==2)
27                    printf("big_endian.\n");
28            else if(un.c[0]==2 && un.c[1]==1)
29                    printf("little_endian.\n");
30            else
31                    printf("unknown .\n");
32            return 0;
33    }
```

说明：在上面的程序中，结构 uts 用来存放系统 CPU 名、操作系统及其版本号(第 13 行)。第 15~19 行调用 uname 函数打印出 CPU 名、操作系统及其版本号。第 26~33 行检测系统字节顺序，并打印输出。以下是在笔者机器上的运行结果：

```
$ ./ex1
i686 -Linux-2.6.22-14-generic:
little_endian.
```

从结果可以看出，系统采用的是 little-endian 方式。

网络协议中的数据采用统一的网络字节顺序，因为只有采用统一的字节顺序，才能在不同类型的机器之间正确地发送和接收数据。Internet 规定的网络字节顺序采用 big-endian 方式。例如 TCP 协议数据段中的 16 位端口号和 32 位 IP 地址就是使用网络字节顺序进行传送的。

Linux 系统提供 4 个库函数来进行字节转换：

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

以上 4 个函数中 h 代表 host，n 代表 network。前两个函数将主机字节顺序转换成网络字节顺序，而后两个函数则刚好相反。编程中，在需要使用网络字节顺序时，应该使用这几个函数来进行转换，绝对不要依赖于具体机器的表示方式。

8.5.4 字节处理函数

套接字地址是多字节数据，不是以空字符结尾的，这和 C 语言中的字符串是不同的。Linux 提供两组函数来处理多字节数据，一组函数以 b(byte)开头，是和 BSD 系统兼容的函数；另一组函数以 mem 开头，是 ANSI C 提供的函数。

以 b 开头的函数有：

```
#include <strings.h>
void bzero(void *s, int n);
void bcopy(const void *src, void *dest, int n);
int bcmp(const void *s1, const void *s2, int n);
```

函数 bzero 将参数 s 指定的内存的前 n 个字节设置为 0。通常用它来将套接字地址清零，如：

```
bzero(&servaddr, sizeof(servaddr));
```

函数 bcopy 从参数 src 指定的内存区域复制指定数目的字节内容到参数 dest 指定的内存区域。函数 bcmp 比较参数 s1 指定的内存区域和参数 s2 指定的内存区域的前 n 个字节内容，如果相同则返回 0，否则返回非 0。

以 mem 开头的函数有：

```
#include <string.h>
void *memset(void *s, int c, size_t n);
void *memcpy(void *dest, const void *src, size_t n);
```

```
int memcmp(const void *s1, const void *s2, size_t n);
```

函数 memset 将参数 s 指定的内存区域的前 n 个字节设置为参数 c 的内容。函数 memcpy 和函数 bcopy 的功能相似，两个函数的差别是：函数 bcopy 能处理参数 src 和参数 dest 所指定的区域有重叠的情况，而函数 memcpy 对这种情况没有定义，这时应该使用函数 bcopy。函数 memcmp 比较参数 s1 和参数 s2 指定区域的前 n 个字节内容，如果相同则返回 0，否则返回非 0。

8.5.5 面向连接的基本套接字函数

本节介绍编写网络程序时使用的基本套接字函数。这一节主要讨论面向连接的 TCP 套接字使用这些函数的情况。

8.5.5.1 socket 函数

函数 socket 创建一个套接字描述符。其定义如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

参数 domain 指定要创建的套接字的协议簇；参数 type 指定套接字类型；参数 protocol 指定使用哪种协议。函数 socket 成功执行时，返回一个正整数，称为套接字描述符，标识这个套接字，否则，返回－1。

Linux 系统的套接字编程接口是一个通用的网络编程接口，它可以访问多种通信协议，如 TCP/IP 协议和 UNIX、域协议等。在创建一个套接字时需要在参数 domain 中指定使用哪种协议簇。参数 domain 的取值如表 8-1 所示。

表 8-1　参数 domain 的取值

值	含　义
AF_UNIX	UNIX 域协议簇，本机的进程间通信时使用
AF_INET	Internet 协议簇
AF_ISO	ISO 协议簇

参数 type 指定套接字类型，取值如表 8-2 所示。

表 8-2　参数 type 的取值

值	含　义
SOCK_STREAM	流套接字，面向连接的和可靠的通信类型
SOCK_DGRAM	数据报套接字，面向非连接的和不可靠的通信类型
SOCK_RAW	原始套接字，只对 Internet 协议有效，可以用来直接访问 IP 协议

参数 orotocol 通常设置为 0，表示使用默认协议，如 Internet 协议簇的流套接字使用 TCP 协议，而数据报套接字使用 UDP 协议。当套接字是原始套接字类型时，需要指定参数 protocol，因为原始套接字对多种协议有效，如 ICMP 和 IGMP 等。

创建一个 TCP 套接字的操作一般如下：

```
sockfd=socket(AF_INET, SOCK_STREAM,0);
if(sockfd<0)
{
 fprintf(stderr, "socket error: %s\n", strerror(errno));
 return -1;
}
```

Linux 系统中创建一个套接字的操作主要是：在内核中创建一个套接字数据结构，然后返回一个套接字描述符标识这个套接字数据结构。这个套接字数据结构包含连接的各种信息，如对方地址、TCP 状态以及发送与接收缓冲区等。TCP 协议根据这个套接字数据结构的内容来控制这条连接。

8.5.5.2 connect 函数

函数 connect 用来与服务器建立一个连接。其定义如下：

```
#include <sys/types.h>
#include <socket.h>
int connect(sockfd, struct sockaddr *servaddr, int addrlen);
```

参数 sockfd 是函数 socket 返回的套接字描述符；参数 servaddr 指定远程服务器的套接字地址，包括服务器的 IP 地址和端口号；参数 addrlen 指定这个套接字地址的长度。函数 connect 成功执行时，返回 0，否则返回 - 1。

在调用函数 connect 之前，客户机需要指定服务器进程的套接字地址。建立一个 TCP 连接的操作一般如下：

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(SERVER_PORT);
if(inet_aton("192.168.0.1",&servaddr.sin_addr)<0)
{
 frpintf(stderr,"inet_aton error\n");
 exit(1);
}
if(connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr))<0)
{
 fprintf(stderr,"connect error");
 exit(1);
```

```
}
```

客户机一般不用指定自己的套接字地址(IP 地址和端口号)，系统会自动从 1024~5000 的端口号范围内为它选择一个未用的端口号，然后以这个端口号和本机的 IP 地址填充这个套接字地址。

客户机调用函数 connect 来主动建立连接。这个函数将启动 TCP 协议的 3 次握手过程。在连接建立之后或发生错误时，函数返回。连接过程中可能有如下几种错误情况：

- 如果客户机 TCP 协议没有接收到对它的 SYN 数据段的确认，函数以错误返回，错误类型为 ETIMEOUT。通常 TCP 协议在发送 SYN 数据段失败之后，会多次发送 SYN 数据段，在所有的发送都告失败之后，函数以错误返回。
- 如果远程 TCP 协议返回一个 RST 数据段，函数立即以错误返问，错误类型为 ECONNREFUSED。当远程机器在 SYN 数据段指定的目的端口号处没有服务器进程在等待连接时，远程机器的 TCP 协议将发送一个 RST 数据段，向客户机报告这个错误。客户机的 TCP 协议在接收到 RST 数据段之后，不再继续发送 SYN 数据段，函数立即以错误返回。
- 如果客户机的 SYN 数据段导致某个路由器产生“目的地不可到达”类型的 ICMP 消息，函数以错误返回，错误类型为 EHOSTUNREACH 或 ENETUNREACH。通常 TCP 协议在接收到这个 ICMP 消息之后，记录这个消息，然后继续几次发送 SYN 数据段，在所有的发送都告失败之后，TCP 协议检查这个 ICMP 消息，函数以错误返回。

如果调用函数 connect 失败，应该用函数 close 关闭这个套接字描述符，不能再次用这个套接字描述符来调用函数 connect。

8.5.5.3 bind 函数

函数 bind 将本地地址与套接字绑定在一起。其定义如下：

```
#inlcude <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

参数 sockfd 是函数 socket 返回的套接字描述符；参数 myaddr 是本地地址；参数 addrlen 是套接字地址结构的长度。函数 bind 成功执行时，返回 0，否则返回 - 1。

服务器和客户机都可以调用函数 bind 来绑定套接字地址，但一般是服务器调用函数 bind 来绑定自己的公认端口号。绑定操作一般如下：

```
bzero(&myaddr, sizeof(myaddr));
myaddr.sin_family=AF_INET;
myaddr.sin_port=htons(PORT);
myaddr.sin_addr.s_addr=htol(INADDR_ANY);
if(bind(sockfd,(struct sockaddr *)&myaddr,sizeof(myaddr))<0)
```

```
{
 frpintf(stderr,"Bind to port %d error \n",PORT);
 exit(1);
}
```

绑定操作一般有表 8-3 所示的几种组合方式。

表 8-3　绑定操作的组合方式

程序类型	IP 地址	端口号	含　义
服务器	INADDR_ANY	非零值	指定服务器的公认端口号
服务器	本地 IP 地址	非零值	指定服务器的 IP 地址和公认端口号
客户机	INADDR_ANY	非零值	指定客户机的连接端口号
客户机	本地 IP 地址	非零值	指定客户机的 IP 地址和连接端口号
客户机	本地 IP 地址	零	指定客户机的 IP 地址

客户机和服务器的其他 IP 地址和端口号组合方式没有什么意义，在这里不讨论。下面详细说明表 8-3 中列出的 5 种方式：

(1) 服务器指定套接字地址的公认端口号，不指定 IP 地址。

服务器调用函数 bind 时，如果设置套接字的 IP 地址为特殊的 INADDR_ANY，表示它愿意接收来自任何网络设备接口的客户机连接。这是服务器最经常使用的绑定方式。

(2) 服务器指定套接字地址的公认端口号和 IP 地址。

服务器调用函数 bind 时，如果设置套接字的 IP 地址为某个本地 IP 地址，表示服务器只接收来自对应于这个 IP 地址的特定网络设备接口的客户机连接。如果这台机器只有一个网络设备接口，这和第(1)种情况是没有区别的，但当这台机器有多个网络设备接口时，可以用这种方式来限制服务器的接收范围。

(3) 客户机指定套接字地址的连接端口号。

在一般情况下，客户机不用指定自己的套接字地址的端口号，当客户机调用函数 connect 进行 TCP 连接时，系统会自动为它选择一个未用的端口号，并且用本地的 IP 地址来填充套接字地址中的相应项。但在有的情况下，客户机需要使用特定端口号，如 Linux 系统中的 rlogin 命令，因为 rlogin 命令需要使用保留端口号，而系统不会为客户机自动分配一个保留端口号，所以需要调用函数 bind 来和一个未用的保留端口号绑定。

(4) 指定客户机的 IP 地址和连接端口号。

表示客户机使用指定的网络设备接口和端口号进行通信。

(5) 指定客户机的 IP 地址。

表示客户机使用指定的网络设备接口进行通信，系统自动为客户机选择一个未用的端口号。一般只有在主机有多个网络设备接口时使用。

在编写客户机程序时，一般不要使用固定的客户机端口号，除非是在必须使用特定端口的情况。固定客户机端口号会带来一些不方便，考虑如下两种情况：

(1) 服务器执行主动关闭操作(如 HTTP 服务器)。

服务器最后进入 TIME_WAIT 状态。当客户机再次与这个服务器进行连接时，仍使用相同的客户机端口号，于是这个连接与前次连接的套接字完全一样，但是因为前次连接处于 TIME_WAIT 状态，并未消失，所以这次连接请求被拒绝，函数 connect 以错误返回。

(2) 客户机执行手动关闭操作(如 FTP 客户机)。

客户机最后进入 TIME_WAIT 状态。当马上再次执行这个客户机程序时，客户机将继续与这个固定客户机端口号绑定，但因为前次连接处于 TIME_WAIT 状态，并未消失，系统会发现这个端口号仍被占用，所以这次绑定操作失败，函数 bind 以错误返回。

8.5.5.4 listen 函数

函数 listen 将一个套接字转换为倾听套接字(listening socket)。其定义如下：

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

参数 sockfd 指定要转换的套接字描述符；参数 backlog 设置请求队列的最大长度。函数 listen 成功执行时，返回 0，否则返回－1。

服务器需要调用函数 listen 将套接字转换成倾听套接字，以便接收客户机请求。函数 listen 的功能有两个：

(1) 函数 socket 创建的套接字是主动套接字，可以用它来进行主动连接(调用函数 connect)，但是不能接收连接请求，而服务器的套接字必须能够接收客户机的请求。函数 listen 将一个尚未连接的主动套接字转换成为一个被动套接字；告诉 TCP 协议，这个套接字可以接收连接请求。

(2) TCP 协议将到达的连接请求排队，函数 listen 的第二个参数指定这个队列的最大长度。

要创建一个倾听套接字，必须首先调用函数 socket 创建一个主动套接字，然后调用函数 bind 将它与服务器套接字地址绑定在一起，最后调用函数 listen 进行转换。这 3 步操作是所有 TCP 服务器所必需的。

下面讨论参数 backlog 的作用，这对于理解套接字建立连接的过程非常重要。TCP 协议为每个倾听套接字维护两个队列：

(1) 未完成连接队列。

每个尚未完成 3 次握手操作的 TCP 连接在这个队列中占有一项。TCP 协议在接收到一个客户机 SYN 数据段之后，在这个队列中创建一个新条目，然后发送对客户机 SYN 数据段的确认和自己的 SYN 数据段(ACK+SYN 数据段)，等待客户机对自己的 SYN 数据段的确认：此时，套接字处于 SYN_RCVD 状态。这个条目将保存在这个队列中，直到客户机返回对 SYN 数据段的确认，或者连接超时。

(2) 完成连接队列。

每个已经完成 3 次握手操作，但尚未被应用程序接收(调用函数 accept)的 TCP 连接，在这个队列中占有一项。当一个在未完成连接队列中的连接接收到对 SYN 数据段的确认之

后，完成 3 次握手操作，TCP 协议将它从未完成连接队列移到完成连接队列中。这个条目将保存在这个队列中，直到应用程序调用函数 accept 来接收它。

参数 backlog 指定倾听套接字的完成连接队列的最大长度，表示这个套接字能够接收的最大数目的未接收(unaccepted)连接。如果当一个客户机的 SYN 数据段到达时，倾听套接字的完成连接队列已经满了，那 TCP 协议将忽略这个 SYN 数据段。对于不能接收的 SYN 数据段，TCP 协议不发送 RST 数据段，原因有两个：

(1) 假设 TCP 协议在未完成队列满时返回 RST 数据段，那么客户机的函数 connect 将马上以错误返回，不再继续发送连接请求。根据这个 RST 数据段，客户机无法知道，究竟是这个端口上没有服务器进程在等待连接，还是在这个端口上等待的服务器的未完成连接队列暂时没有空间。

(2) 完成队列满的情况是暂时的：经过一段时间之后，应用程序可能调用函数 accept 从这个完成队列中接收已经建立的连接，于是完成队列中出现新的空间。客户机 TCP 协议在超时之后，继续几次发送 SYN 数据段。如果在这几次发送过程中，完成连接队列中出现新的空间，那么 TCP 协议将接收这个连接请求，继续正常的 3 次握手操作。如果在这几次发送过程中，完成连接队列中都没有空间，客户机将放弃发送。

8.5.5.5 accept 函数

函数 accept 从倾听套接字的完成连接队列中接收一个连接。如果完成连接队列为空，那么这个进程睡眠。其定义如下：

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

参数 sockfd 指定套接字描述符；参数 addr 为指向一个 Internet 套接字地址结构的指针；参数 addrlen 为指向一个整型变量的指针。函数 accept 成功执行时，返回 3 个结果：函数返回值为一个新的套接字描述符，标识这个接收的连接；参数 addr 指向的结构变量中存储客户机地址；参数 addrlen 指向的整型变量中存储客户机地址的长度。如果对客户机的地址和长度都不感兴趣，可以将参数 addr 和 addrlen 设置为 NULL。函数 accept 执行失败时，返回－1。

函数 accept 从倾听套接字的完成连接队列中接收一个已经建立起来的 TCP 连接，因为倾听套接字是专为接收客户机连接请求，完成 3 次握手操作而用的，所以 TCP 协议不能使用倾听套接字描述符来标识这个连接，于是 TCP 协议创建一个新的套接字来标识这个要接收的连接，并将它的描述符返回给应用程序。现在有两个套接字，一个是调用函数 accept 时使用的倾听套接字，另一个是函数 accept 返回的连接套接字(connected socket)。这两个套接字的作用是完全不同的：一个服务器进程通常只需创建一个倾听套接字，在服务器进程的整个活动期间，用它来接收所有客户机的连接请求，在服务器进程终止前关闭这个倾听套接字；而对于每个接收的(accepted)连接，TCP 协议都创建一个新的连接套接字，来标识这个连接，服务器使用这个连接套接字与客户机进行通信操作，当服务器处理完这个客户机请求时，关闭这个连接套接字。

当函数 accept 阻塞等待已经建立的连接时，如果进程捕获到信号，那么函数将以错误返回，错误类型为 EINTR。对于这种错误，一般重新调用函数 accept 来接收连接。

8.5.5.6 close 函数

函数 close 关闭一个套接字描述符。套接字描述符的 close 操作与文件描述符的 close 操作类似。其定义如下：

```
#include <unistd.h>
int close(sockfd);
```

参数 sockfd 指定要关闭的套接字描述符。函数 close 成功执行时，返回 0，否则返回－1。

套接字描述符的 close 操作和文件描述符的 close 操作一样：函数 close 将套接字描述符的引用计数减 1，如果描述符的引用计数大于 0，表示示还有进程引用这个描述符，函数 close 正常返回；如果描述符的引用计数变为 0，则表示再没有进程引用这个描述符，于是启动清除套接字描述符的操作，函数 close 立即正常返回。清除套接字描述符的操作是：将这个套接字描述符标记为关闭状态，然后立即返回进程。调用了函数 close 之后，进程将不再能够访问这个套接字，但是这不表示 TCP 协议删除了这个套接字。TCP 协议将继续使用这个套接字，将尚未发送的数据传递到对方，然后发送 FIN 数据段，执行关闭操作，一直等到这个 TCP 连接完全关闭之后，TCP 协议才删除这个套接字。

8.5.5.7 read 和 write 函数

函数 read 和 write 从套接字读和写数据。其定义如下：

```
int read(int fd, char *buf, int len);
int write(int fd, char *buf ,int len);
```

参数 fd 指定读写操作的套接字描述符；函数 read 的参数 buf 指定接收数据缓冲区，函数 write 的参数 buf 指定发送数据缓冲区；参数 len 指定接收或发送的数据量大小。函数 read 成功执行时，返回读到的数据量大小，否则返回－1。函数 write 成功执行，返回写入的数据量大小，否则返回－1。

前面介绍了基本的套接字函数，下面就这些函数举例说明其应用。

例 8-2 编写一个客户机/服务器程序，其中客户机使用流套接字向服务器请求日期和时间，服务器在收到请求后，回答请求并显示出客户的地址。

首先看服务器程序：

```
1    /*ex2serv.c*/
2    #include <time.h>
3    #include <stdio.h>
4    #include <stdlib.h>
5    #include <sys/socket.h>
6    #include <netinet/in.h>
```

```
7    #include <netdb.h>
8
9    #define LISTENQ      5
10   #define     MAXLINE     512
11
12   int main()
13   {
14       int listenfd, connfd;
15       socklen_t   len;
16       struct sockaddr_in servaddr, cliaddr;
17       char  buff[MAXLINE];
18       time_t       ticks;
19       listenfd=socket(AF_INET, SOCK_STREAM,0);
20       if(listenfd<0)
21       {
22           printf("Socket created failed.\n");
23           return -1;
24       }
25       servaddr.sin_family=AF_INET;
26       servaddr.sin_port=htons(6666);
27       servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
28       if(bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr))<0)
29       {
30           printf("bind failed.\n");
31           return -1;
32       }
33       printf("listening....\n");
34       listen(listenfd, LISTENQ);
35       while(1)
36       {
37           len=sizeof(cliaddr);
38           connfd=accept(listenfd,(struct sockaddr *)&cliaddr, &len);
39           printf("connect from %s, port %d
              \n",inet_ntoa(cliaddr.sin_addr.s_addr),ntohs(cliaddr.sin_port));
40           ticks=time(NULL);
41           sprintf(buff,"% .24s \r \n",ctime(&ticks));
42           write(connfd,buff,strlen(buff));
43           close(connfd);
44       }
45   }
```

说明：在上面的面向连接的套接字的程序中，服务器首先调用 socket 函数创建一个流套接字(第 19~24 行)，随后用本机地址和 6666 端口号填充 sockaddr_in 结构(第 25~27 行)，

然后调用 bind 函数把创建的套接字绑定在这个地址上(第 28~32 行)，接着服务器就调用 listen 函数监听连接(第 34 行)，当有连接进入时，accept 函数接收所引入的请求，打印出客户机的地址(第 38~39 行)，然后格式化服务器本地时间并用 write 函数回答客户机请求，最后关闭套接字(第 40~43 行)。

下面再看一下客户机程序：

```
1   /*ex2cli.c*/
2   #include <stdio.h>
3   #include <sys/socket.h>
4   #include <netinet/in.h>
5   #include <netdb.h>
6   #define     MAXBUFFSIZE 256
7   #define     PORT 6666
8   #define HOST_ADDR "127.0.0.1"
9   int main(int argc, char *argv[])
10  {
11          int sockfd,n;
12          char recvbuff[MAXBUFFSIZE];
13          struct sockaddr_in      servaddr;
14          sockfd=socket(AF_INET,SOCK_STREAM,0);
15          if(sockfd<0)
16          {
17                  printf("Socket created failed.\n");
18                  return -1;
19          }
20
21          servaddr.sin_family=AF_INET;
22          servaddr.sin_port=htons(6666);
23          servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
24          printf("connecting...\n");
25          if(connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr))<0)
26          {
27                  printf("Connect server failed.\n");
28                  return -1;
29          }
30          while((n=read(sockfd,recvbuff,MAXBUFFSIZE))>0)
31          {
32                  recvbuff[n]=0;
33                  fputs(recvbuff,stdout);
34          }
35          if(n<0)
36          {
37                  printf("Read failed!\n");
```

```
38                  return -2;
39            }
40            return 0;
41      }
```

说明：在上面的程序中，客户机首先调用 socket 函数创建一个套接字(第 14~19 行)，随后用服务器地址(在本例中服务器地址就是本机地址)和 6666 端口号填充 sockaddr_in 结构(第 21~23 行)，然后调用 connect 函数试图连接服务器(第 25~29 行)，如果连接成功调用 read 接收从服务器发送过来的日期信息，并打印输出，随后程序退出(第 30~40 行)。

程序执行结果如下：

```
$ ./ex2serv &                                          /*后台执行服务器程序 */
[3] 14587
[2]    已终止                    ./ex2serv
lxy@lxy-desktop:~/test1/chapter9$ listening.... /*服务器开始监听 */
./ex2cli                                               /* 执行客户机程序 */
connecting...
connect from 127.0.0.1, port 42702                     /*服务器输出 */
 ri May 16 23:16:57 2008                        /*客户机输出 */
```

8.5.6 其他套接字操作函数

除了以上提到的基本套接字操作函数外，下列函数也经常用于面向连接的通信过程和数据报通信过程。

8.5.6.1 getsockname 和 getpeername

这两个函数一个用于返回本地套接字地址，一个用于返回与一个套接字相连的对等套接字地址。

函数 getsockname 用来获取一个本地套接字地址。它的原型如下：

```
#include <sys/socket.h>
int getsockname(int socket, socket sockaddr * address, socklen_t *address_len);
```

getsockname 函数获取由描述符 socket 给出的套接字的本地捆绑名。当它调用成功时返回 0，并存储 socket 的地址于 address 参数所指的 sockaddr 结构对象中，存储其地址长度于 address_len 所指对象中。

如果地址的实际长度大于 address 所指对象的长度，存储的地址将被截断；如果 socket 还没有捆绑地址，存储在 address 所指对象中的值将是未定义的。

getsockname 调用失败返回 –1。

存储在 address 所指对象中的地址的格式依赖于该套接字的通信域。对于给定的通信域，套接字地址的长度通常是固定的，因此，一般可以确切地知道需要多少空间，并提供

实际需要的存储空间。通常的做法是用与套接字通信域相匹配的数据类型为 address 所指对象分配空间，然后强制其地址转“struct socket *”并传送给 getsockname。

在下述情况下需要调用 getsockname 函数：

(1) 对于没有使用 bind 捆绑地址至套按字的客户进程，在它成功调用 connect 之后，getsockname 可以返回内核指定给该套接字的本地地址(如 IP 地址和端口号等)。

(2) 当用 0 端口号(告诉内核选择本地端口号)调用 bind 之后，getsockname 可以返回内核指定给该套接字的本地端口号。

(3) getsockname 可以获得一个套接字的地址簇。

(4) 服务进程在接收了客户的连接之后(成功调用 accept 之后)，以 accept 返回的描述符调用 getsockname 可以获得指定给该连接的套接字地址，这个套接字是实际连接的套接字，而不是侦听套接字。

例 8-3 下面的程序说明了 getsockname 的用法，它打印出用默认地址命名的套接字的实际地址。

```
1      /* ex3.c*/
2      #include <sys/socket.h>
3      #include <sys/time.h>
4      #include <netinet/in.h>
5      #include <netdb.h>
6      #include <stdio.h>
7
8      int main()
9      {
10             int sockfd;
11             socklen_t   len;
12             struct sockaddr_in addr, raddr;
13             sockfd=socket(AF_INET,SOCK_STREAM,0);
14             if(sockfd<0)
15             {
16                     printf("create socket failed!\n");
17                     return -1;
18             }
19             addr.sin_family=AF_INET;
20             addr.sin_port=htons(0);
21             addr.sin_addr.s_addr=htonl(INADDR_ANY);
22             if(bind(sockfd,(struct sockaddr *)&addr,sizeof(addr))<0)
23             {
24                     printf("bind socket failed!\n");
25                     return -2;
26             }
27             len=sizeof(raddr);
```

```
28          if(getsockname(sockfd,(struct sockaddr *)&raddr,&len)<0)
29          {
30                printf("getsockname failed!\n");
31                return -3;
32          }
33          printf("bound name= %s, port =%d \n",
             ntohl(raddr.sin_addr.s_addr),ntohs(raddr.sin_port));
34          return 0;
35    }
```

说明：在上面的程序中，首先创建一个套接字(第 13 行)，然后命名该套接字(第 19~21 行)，随后调用 getsockname 函数查看套接字相连的名字和端口号，并打印输出结果(第 28~33 行)。程序执行结果如下：

```
$ ./ex3
bound name= (null), port =47140
```

函数 getpeername 用于获取一个套接字的远程对等套接字的地址。它的原型如下：

```
#include <sys/socket.h>
int getpeername(int socket, struct sockaddr *address, socklen_t *address_len);
```

getpeername 返回与 socket 连接的那个套接字的地址。它存储这个地址于 address 所指对象，存储该地址的长度于 address_len 所指对象。

如果地址的实际长度大于 address 提供的长度，存储的地址将被截断。

该函数成功时返回值 0，出错时返回 − 1。

例 8-4 服务进程的套接字是被动套接字，它可以接收连接，但无法选择连接的对象。但有时为了拒绝那些不希望的连接，需要知道连接从何而来，并且当不希望与那个进程交谈时关闭该连接。下面的程序说明了如何用 getpeername 来查看对等套接字，并在不希望与之交谈时关闭与它的连接。

```
1     /* ex4.c */
2     int check_peer(int sockfd, in_addr_t *refuselist)
3     {
4           socklen_t   len;
5           struct sockaddr_in addr, raddr;
6           in_addr_t s_addr;
7           len=sizeof(raddr);
8           if(getpeername(sockfd,(struct sockaddr *)raddr, &len)<0)
9           {
10                printf("getpeername with socket %d failed.\n",sockfd);
11                return -1;
12          }
```

```
13          s_addr=raddr.sin_addr.s_addr;
14          ap=refuselist;
15          for(; ap!=0;ap++)
16          {
17              close(sockfd);
18              return -1;
19          }
20          return 0;
21      }
```

说明：在上面的程序中，调用 getpeername 函数查看与服务进程连接的 socket(第 8~12 行)，如果 socket 在拒绝地址列表中，则拒绝该连接(第 13~19 行)。

尽管从 accept 的返回参数中也能得到对等套接字的地址，但是当服务程序是由调 accept 的进程通过 fork 和 exec 而执行时，由于 accept 返回参数的存储空间位于父进程中，因此经 exec 后它将不复存在。在这种情况下，getpeername 是唯一能获得对等套接字地址的方法。

8.5.6.2 send 和 recv

套接字之间一旦建立了连接，就可以开始传送数据。在前面的例子中，我们使用的是标准读、写函数 read 和 write，这一节介绍另外两个函数 recv 和 send。

recv 和 send 函数类似于标准的 read 和 write 函数，但它们只能用于套接字，并且还需要一个另外的参数，此参数指明控制套接字特殊传输方式的各种标志。例如，我们可以指明 MSG_OOB 标志读写带外数据，指明 MSG_PEEK 标志查看当前数据，或指明 MSG_DONTROUTE 标志来控制包含在输出中的路由信息。

send 函数用于向已连接的套接字发送数据。它的原型如下：

```
#include <sys/socket.h>
ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

该函数启动从 socket 指定的套接字传送一条消息至对等套接字。它的前 3 个参数与 write 函数相同。最后一个参数 flags 指明消息传送的类型，它的值要么为 0(此时这个函数等价于 write)，要么由表 8-4 所示标志按位逻辑或操作形成。

表 8-4 参数 flags 的标志位

值	含　义
MSG_OOB	导致 send 发送的数据成为带外数据
MSG_DONTROUTE	不在消息中包含路由信息

带外数据是流套接字特有的。在流套接字上传送数据时，数据按它们写出的顺序传送。因为接收进程必须依次读套接字上的当前数据，因此，当出现一个紧急情况时，没有办法立即通知接收进程。带外数据正用于解决这一问题。带外数据在正常的数据流之外发送，其效果相当于越过套接字上所有等待读的数据。当它到达接收进程时，接收进程会收到一

个信号，从而进程可以立即处理这个数据。

通常只有诊断或路由程序才对 MSG_DONTROUTE 标志感兴趣，一般程序并不关心此标志。

该函数调用成功返回实际传送的字节个数，失败时返回－1。不过要注意，一个成功的返回值仅仅指出已正确地将消息发送出去而已，并不一定意味着该消息已正确地被接收。返回值为－1 仅指出本地检测到的错误。

同 write 一样，send 函数也是阻塞的，如果套接字不能立即传送数据，send 将等待直至数据被传送完之后才返回。

对于非阻塞套接字，如果数据不能立即传送，send 将设置 SEWOULDBLOCK 而失败返回。

send 所传送的实际数据长度可能小于参数 length 指定的长度。注意，如果 send 返回的值不等于 length，继续发送剩余的数据是其责任。通常，当数据包的大小小于 1KB 时，send 能够完整地发送所有数据。

如果要传送的消息太长而不能由底层的协议传送，send 将失败并且没有传送数据。如果该套接字曾连接过但其连接已断开，则使用 send 或 write 均会得到 SIGPIPE 信号。

recv 函数用于从已连接的套接字接收消息。它的原型如下：

```
#include <sys/socket.h>
ssize_t recv(int socket, void * buffer, size_t length, int flags);
```

该函数类似于 read，它的前三个参数也与 read 相同，但还有一额外的参数 flags。该参数指明消息接收的类型，它的值要么为 0(此时这个函数等价于 read)，要么由表 8-5 所示标志按位逻辑或操作而形成。

表 8-5 参数 flags 的标志位

值	含 义
MSG_OOB	读带外数据
MSG_PEEK	查看套接字上的数据而不实际读出它们，即尽管 buffer 所指对象中填入了所请求的数据，随后的 read 或 recv 将读到相同的数据
MSG_WAITALL	请求函数阻塞直至所请求的全部数据都已接收到。不过，在下述情况下，尽管指明了 MSG_WAITALL 标志，recv 接收到的数据仍然可能小于要求的数据的大小： 1. 出现信号 2. 连接被中断 3. 指明了 MSG_PEEK 4. 套接字出错

recv 调用成功返回读到 buffer 所指缓冲区中的数据的字节长度；如果没有消息可接收

并且对等套接字已执行了 shutdown，将返回 0，否则返回 -1。

同 read 一样，recv 也是阻塞的。如果在套接字上没有可读的消息，recv 将等待直至有消息到达。当套接字设置了非阻塞标志 O_NONBLOCK 且没有数据可读时，recv 立即返回而不是等待。

read 和 write 通常用来读写套接字上的普通数据，当需要发送或接收特殊数据，如带外数据时，就必须使用 send 和 recv 才能做到。

例 8-5 下面的程序是一个简单的发送带外数据的程序。

```
1    /* ex5.c */
2    #include <stdio.h>
3    #include <sys/socket.h>
4    #include <netinet/in.h>
5    #include <netdb.h>
6
7    #define     PORT 6666
8    #define HOST_ADDR "127.0.0.1"
9    int main()
10   {
11           int sockfd,n;
12           struct sockaddr_in      servaddr;
13           sockfd=socket(AF_INET,SOCK_STREAM,0);
14           if(sockfd<0)
15           {
16                   printf("Socket created failed.\n");
17                   return -1;
18           }
19           servaddr.sin_family=AF_INET;
20           servaddr.sin_port=htons(6666);
21           servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
22           printf("connecting...\n");
23           if(connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr))<0)
24           {
25                   printf("Connect    Error.\n");
26                   return -1;
27           }
28           write(sockfd,"123",3);
29           printf("wrote 3 byte of normal data. \n");
30           sleep(1);
31           send(sockfd,"a",1,MSG_OOB);
32           printf("wrote 1 byte of OOB data.\n");
33           sleep(1);
34           write(sockfd,"56",2);
```

```
35          printf("wrote 2 byte of normal data \n");
36          sleep(1);
37          send(sockfd,"b",1,MSG_OOB);
38          printf("wrote 1 byte of OOB data\n");
39          sleep(1);
40          write(sockfd,"89",2);
41          printf("wrote 2 byte of normal data \n");
42          sleep(1);
43          return 0;
44      }
```

说明：上面的程序首先创建一个套接字(第 13 行)，并命名该套接字(第 19~21 行)，然后连接服务程序(第 23~27 行)，连接成功后，总共发出 9B 的数据，其中夹杂着 3 个带外数据，并且每一个输出操作之后睡眠 1S。睡眠的目的是为了使每一个 write 或 send 作为单个 TCP 段被发送和接收(第 28~42 行)。

接收带外数据的情形则要复杂一些，因为必须注意带外数据的通知。下面是使用 SIGURG 信号接收带外数据的例子。

例 8-6　接收带外数据的例子。

```
1       /*ex6.c*/
2       #include <sys/socket.h>
3       #include <netinet/in.h>
4       #include <netdb.h>
5       #include <stdio.h>
6       #include <signal.h>
7       #include <fcntl.h>
8       #include <unistd.h>
9       #define   LISTENQ    5
10
11
12      int listenfd, connfd;
13      void sig_urg(int signo);
14      int main()
15      {
16          int n;
17          char buff[100];
18          socklen_t   len;
19          struct sockaddr_in       servaddr, cliaddr;
20          struct sigaction action;
21          action.sa_handler=sig_urg;
22          sigemptyset(&action.sa_mask);
23          action.sa_flags=0;
24          listenfd=socket(AF_INET, SOCK_STREAM,0);
```

```
25          if(listenfd<0)
26          {
27                  printf("Socket created failed.\n");
28                  return -1;
29          }
30          servaddr.sin_family=AF_INET;
31          servaddr.sin_port=htons(6666);
32          servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
33          if(bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr))<0)
34          {
35                  printf("bind failed.\n");
36                  return -1;
37          }
38          printf("listening....\n");
39          listen(listenfd, LISTENQ);
40          len=sizeof(cliaddr);
41          connfd=accept(listenfd,(struct sockaddr *)&cliaddr, &len);
42          if(sigaction(SIGURG,&action, NULL)==-1)
43          {
44                  printf("Couldn't register signal handler.\n");
45                  return -2;
46          }
47          fcntl(connfd,F_SETOWN,getpid());
48          while(1)
49                  {
50                    if((n=read(connfd,buff,sizeof(buff)))==0)
51                          {
52                                  printf("received EOF \n");
53                                  return 0;
54                          }
55                  buff[n]=0;
56                  printf("read %d bytes: %s \n",n,buff);
57                  }
58  }
59  
60  void sig_urg(int signo)
61  {
62          int n;
63          char buff[100];
64          printf("SIGURG received \n");
65          n=recv(connfd,buff,sizeof(buff),MSG_OOB);
66          buff[n]=0;
67          printf("read %d OOB byte: %s\n",n,buff);
```

```
68    }
```

说明：上面的接收程序使用 SIGURG 信号来获得带外数据的通知。当一套接字发现有一带外数据在途中时，会发送 SIGURG 信号给此套接字的拥有进程或进程组；为了捕获该信号，首先要建立 SIGURG 的句柄并且调用 fcntl 设置连接套接字的拥有者(第 21~47 行)。

while 循环中利用 read 读客户发来的正常数据。当客户终止其连接时，该接收程序将因读到 EOF 而终止(第 48~57 行)。

SIGURG 信号句柄函数 sig_urg 专门读带外数据，当捕获到 SIGURG 信号时，进程执行该函数。为了读带外数据，要用具有 MSG_OOB 标志的 recv 函数，普通的读操作不能读带外数据，只能读普通数据(第 60~68 行)。

先在一个终端窗口中运行程序 6，然后在另一个终端窗口中运行程序 5，结果如下：

```
$ ./ex6
listening....
read 3 bytes: 123
SIGURG received
read 1 OOB byte: a
read 2 bytes: 56
SIGURG received
read 1 OOB byte: b
read 2 bytes: 89
received EOF
$ ./ex5
connecting...
wrote 3 byte of normal data.
wrote 1 byte of OOB data.
wrote 2 byte of normal data
wrote 1 byte of OOB data
wrote 2 byte of normal data
```

结果如预料的那样，发送程序发送的每一个带外数据都对接收程序生成了一个 SIGURG 信号，这个信号使得接收程序读到了一个字节的带外数据。

8.5.7 数据报套接字操作

由于底层的协议不同，数据报套接字与流套接字有一些基本的差异。数据报套接字是 UDP 协议，UDP 是无连接、不可靠的数据报协议。图 8-5 说明了典型的数据报套接字客户机/服务器程序所使用的函数和通信过程。在这种通信方式中，客户不与服务建立连接，它只是通过 sendto 向服务程序发送数据报，sendto 函数本身要求一个地址参数给出服务程序的地址。同样，服务程序也不接收来自客户的连接，它只是调用 recvfrom 函数，这个函数等待来自某个客户的数据，并随接收到的数据报一起返回客户的地址，服务程序由此可以

回应客户。使用数据报套接字，可以将数据集中为一个包，为每个包单独地指定目的地址，并且每个包独立地进行通信。

这一节介绍数据报套接字操作使用的两个函数 sendto 和 recvfrom，给出数据报套接字客户机/服务器程序的例子，用数据报套接字实现广播通信的方法和例子。对于数据报套接字，不允许使用 listen 和 accept 函数。

sendto 和 recvfrom

在数据报套接字上发送和接收数据的正常方法是使用 sendto 和 recvfrom 函数。sendto 向数据报套接字发送数据包，recvfrom 从数据报套接字读数据包，同时报告该包从何而来。它们的原型如下：

```
#include <sys/socket.h>
int recvfrom(int socket, void *buffer, size_t size, int flags, struct sockaddr *from, size_t * addrlen);
int sendto(int socket, void *buffer, size_t size, int flags, struct sockaddr *to, size_t addrlen);
```

这两个函数的前 3 个参数 socket、buffer、size 与 read 和 write 的参数相同，它们分别为套接字描述符、指向读写缓冲区的指针以及读写的字节数。对于 recvfrom，如果所接收包的实际长度大于 size 字节，则得到该包的前 size 个字节，而包的剩余部分丢失，无法再读到此包的剩余部分。因此，当采用包协议时，必须总是知道包的预期长度。

参数 flags 的解释同 send 和 recv 的情形。我们只讨论简单数据报客户机 / 服务器的例子，在这种情况下，不需要使用这个标志，它的值总是为 0。

recvfrom 的参数 from 和 addrlen 类似于 accept 的最后两个参数：在函数返回时，它们给出的套接字地址结构告之是谁发送的数据报。如果对这一信息不感兴趣，可指定 from 为一空指针，不过要注意，此时参数 addrlen 也必须为空指针。

sendto 的最后两个参数类似于 connect；当发送数据报时，要在此套接字地址结构中填入协议地址以指明数据报发送给谁。注意，sendto 的最后一个参数是整数值，而 recvfrom 的最后一个参数是指向整数的指针。

这两个函数的返回值与错误条件也与 send 和 recv 的相同，函数的返回值是实际读写的字节数。我们不能完全依赖系统来检测和报告错误，因为最常见的错误是包被丢失，或是在指定的地址没有对等套接字接收消息，对于这类错误，要由程序本身来检测。

这两个函数都可用于流套接字，尽管很少这样使用。另外，如果不必检查数据报是由谁发送的，可以使用普通的 recv 替代 recvfrom，当不想指定 flags 时甚至还可用 read。

下面给出一个数据报客户机/服务器程序例子。

例 8-7 数据报通信服务器程序。

```
1    /*ex7.c*/
2    #include <sys/types.h>
3    #include <sys/socket.h>
4    #include <netinet/in.h>
```

```
5    #include <stdio.h>
6
7    #define MAXMSG 1024
8
9    int main()
10   {
11           int sockfd,size,nbytes;
12           struct sockaddr_in addr;
13           char message[MAXMSG];
14           sockfd=socket(AF_INET,SOCK_DGRAM,0);
15           if(sockfd<0)
16           {
17                   printf("Socket created failed.\n");
18                   return -1;
19           }
20           bzero(&addr,sizeof(addr));
21           addr.sin_family=AF_INET;
22           addr.sin_port=htons(6666);
23           addr.sin_addr.s_addr=htonl(INADDR_ANY);
24           if(bind(sockfd, (struct sockaddr *)&addr, sizeof(addr))<0)
25           {
26                   printf("bind failed.\n");
27                   return -1;
28           }
29           while(1)
30           {
31                   size=sizeof(addr);
32                   nbytes=recvfrom(sockfd,message,MAXMSG,0,(struct sockaddr *)&addr,&size);
33                   if(nbytes<0)
34                   {
35                   printf("recvfrom(server) failed.\n");
36                   return -1;
37                   }
38                   printf("Server got message: %s \n",message);
39                   nbytes=sendto(sockfd,message,nbytes,0,(struct sockaddr*)&addr,size);
40                   if(nbytes<0)
41                   {
42                   printf("sendto(server) failed.\n");
43                   return -1;
44                   }
45           }
46   }
```

说明：服务程序创建一个数据报套接字(第 14~19 行)，并给它捆绑一个通配名地址(第 20~28 行)，然后从它接收数据，收到数据后，打印输出收到的消息，并将消息再返回给发送者(第 31~44 行)。

下面再看客户机程序。

例 8-8 数据报通信客户机程序。

```
1     /*ex8.c*/
2     #include <stdio.h>
3     #include <sys/socket.h>
4     #include <netinet/in.h>
5     #include <netdb.h>
6
7     #define MAXMSG 512
8
9     int main()
10    {
11            int sockfd,n;
12            char recvbuff[MAXMSG],sndbuff[MAXMSG];
13            struct sockaddr_in        servaddr;
14            sockfd=socket(AF_INET,SOCK_DGRAM,0);
15            if(sockfd<0)
16            {
17                    printf("Socket created failed.\n");
18                    return -1;
19            }
20
21            servaddr.sin_family=AF_INET;
22            servaddr.sin_port=htons(6666);
23            servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
24            while(fgets(sndbuff,MAXMSG,stdin)!=NULL)
25            {
26                    if(sendto(sockfd,sndbuff,sizeof(sndbuff),0,&servaddr,sizeof(servaddr))<0)
27                    {
28                            printf("(client)sending error.\n");
29                            return -1;
30                    }
31                    if((n=recvfrom(sockfd,recvbuff,MAXMSG,0,NULL,NULL))<0)
32                    {
33                            printf("(client)receiving error.\n");
34                            return -1;
35                    }
36                    recvbuff[n]=0;
37                    printf("Client received message: %s",recvbuff);
```

```
38          }
39          close(sockfd);
40          return 0;
41      }
```

说明：在上面的程序中，首先创建数据报套接字(第 14~19 行)，并用服务器地址和端口号形成发送目的地套接字地址(第 21~23 行)。在 while 循环中，首先从终端接受输入的消息，然后发送该消息并从服务器接收显示服务器返回的回答。最后关闭套接字描述符，退出程序(第 24~40 行)。

下面是这个例子的运行实例。运行时先在一个窗口终端启动接收程序 ex7：

```
$ ./ex7&
[1] 5773
```

再开启另一个窗口终端，运行发送程序 ex8，输入消息，得到如下结果：

```
$ ./ex8
hello,how are you?
Client received message: hello,how are you?
Fine.Thank you.And you?
Client received message: Fine.Thank you.And you?
I am fine too
Client received message: I am fine too
```

然后再开启一个终端窗口，运行程序 ex8。注意，此时前面运行的 2 个程序都没有停止：

```
$ ./ex8
Today is Sunday.
Client received message: Today is Sunday.
```

第一个终端窗口的显示为：

```
Server got message: Today is Sunday.
```

输入 Ctrl-d，程序由于读到文件结束符而结束。现在回到前面运行客户程序的终端完成如下操作：

```
$ ./ex8
See you next time,bye!
Client received message: See you next time,bye!
```

与此同时，在运行服务程序的窗口终端上将出现如下信息：

```
$ Server got message hello,how are you?
```

```
Server got message Fine.Thank you.And you?

Server got message I am fine too

Server got message Today is Sunday

Server got message See you next tim,bye!
```

最后，注意不要忘记用 kill 命令终止服务程序 ex7，因为这个程序自己决不会终止。从服务程序的输出结果可以看出，它如所料地收到了这两个客户发来的消息，并且接收消息的顺序是数据报到达的顺序。

8.6 小　结

网络功能强大是 Linux 最著名的特点之一，之所以有那么多的网络服务器使用 Linux 作为操作平台，是因为 Linux 提供对各种网络服务广泛的支持。

本章中我们介绍了 Linux 对不同的网络协议的支持，其中主要是对 TCP / IP 的支持。在介绍了 TCP/IP 的基础上，介绍了 Linux 的网络编程接口——BSD 套接字接口和客户机/服务器程序模式，最后介绍了面向连接的数据流套接字和无连接的数据报套接字的编程一般步骤，并给出了详细的例子。读者可以在此基础上进行更深入的学习。

习　题

一、填空题

1. TCP/IP 协议参考模型共分______层，它们分别是______、______、______、______。

2. 利用套接字进行通信的进程采用______模式。

3. Linux 支持伯克利(BSD)风格的套接字编程。它同时支持______和______的套接字。

4. ______套接字定义了一种可靠的面向连接的服务，实现了无差错、无重复的顺序数据传输。______套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，

5. 在计算机内存中有 2 种存储整数的方式，低位字节存储在这个整数的开始地址位置，是______方式，高位字节存储在开始地址位置是______方式。

二、选择题。

1. 下列不属于应用层协议的是______。

(A) FTP　(B) HTTP　(C) TCP　(D) DNS

2. 考虑一个整数 1234，在 little-endian 方式下，在内存中存放方式是______。

(A) 12 34　(B) 34 12　(C) 12 43 (D) 43 21

3. 用于返回本地套接字地址的函数是______。

(A) socket　(B) getsockname　(C)　getpeername　(D) getsocket

4. 要创建一个倾听套接字，必须首先调用函数______创建一个主动套接字，然后调用函数______将它与服务器套接字地址绑定在一起，最后调用函数______。

(A) bind　(B) socket　(C) create　(D) listen

5. 在数据报套接字上发送和接收数据的正常方法是使用______和______函数。

(A) send　(B) sendto　(C)recv　(D)recvfrom

三、上机题

1. 编写一个程序，测试一下本机存储整数的方式是 little-endia 方式还是 big-endian 方式。

2. 编写一个数据流客户机/服务器程序，服务器程序在与客户机程序建立连接后，返回服务器的时间，客户机程序在接收后将时间显示出来。

3. 编写一个数据流客户机/服务器程序，其中客户机接受键盘输入并发送到服务器，服务器把输入内容显示出来。

4. 编写一个数据报客户机/服务器程序，客户机向服务器发送“Hello World！”字符串，服务器在收到字符串后，将客户机 IP 地址、端口号以及字符串内容显示出来。

CHAPTER 9

数据库编程

对于计算机应用来讲，科学计算进入到数据处理是一个划时代的转折。这个转折使计算机从少数科学家手中的珍品转变成为科技人员和管理人员甚至普通用户的得力助手和有力工具。

几乎所有的应用程序都要存储数据。数据的范围很广，从少量的程序运行参数到巨大的复杂数据库，比如全国人口普查信息、一个公司一年的营业收入等。数据处理是指对各种形式的数据进行收集、存储、加工和传播的一系列活动的总和，其目的是从大量的原始数据中抽取并推导出对人们有价值的信息作为行动和决策的依据，并借助计算机科学地保存和管理大量复杂的数据，以充分而方便地利用信息资源。

数据处理的中心问题是数据管理。数据管理是指对数据的收集、存储、检索和维护。数据管理随着计算机硬件和软件的发展而不断发展。40 多年来数据管理经历了如下三个阶段：人工管理阶段、文件系统阶段和数据库系统阶段。

数据库系统是当代计算机系统的重要组成部分。数据库技术所研究的问题就是如何科学地组织和存储数据，如何高效地获取、更新和加工数据，并保证数据的安全性、可靠性和持久性。

Linux 的数据库功能很强，它常被用于数据库服务器端，因此有必要了解 Linux 数据库开发方面的知识。本章将以 MySQL 为例讲述数据库开发的一些基础知识，让读者对数据库的开发有一个基本认识。

在介绍 MySQL 数据库开发之前，先介绍一些数据库的基础知识。

9.1　数据库基本概念

在系统地了解数据库知识之前，应先了解和熟悉数据库的一些最基本的术语和概念：

数据、数据库、数据库管理系统、数据库语言和数据库系统。

9.1.1 数据与数据库

1. 数据(Data)

数据是数据库中存储的基本对象。在计算机中可表示数据的种类很多，文字、图形、图像、声音都可以数字化。为了了解世界、交流信息，人们需要通过计算机来描述、存储和处理这些表现形式多样和内容复杂的数据。

可以对数据作如下定义：描述事物的符号记录称为“数据”。描写事物的符号可以是数字，也可以是文字、图形、图像和声音等，即有多种表现形式，但它们都是经过数字化后存入计算机的。

2. 数据库(DataBase，缩写为 DB)

数据库可以直观地理解为存放数据的仓库，这个仓库在计算机的大容量存储器上(如硬盘等)。数据必须按照一定的格式存放，以便于查找。可以认为数据库是被长期存放在计算机内、有组织的、可以表现为多种形式的可共享的数据集合。数据库技术使得数据能够按一定格式组织、描述和存储，而且具有较小的冗余度、较高的数据独立性和易扩展性，并可为多个用户所共享。

9.1.2 数据库管理系统

数据库管理系统(Data Base Management System，缩写为 DBMS)是位于用户与操作系统之上的一层数据管理软件。数据库管理系统是为数据库的建立、使用和维护而配置的软件。它建立在操作系统的基础上，对数据库进行统一的管理和控制。用户使用的各种数据库命令以及应用程序的执行，都要通过数据库管理系统。数据库管理系统还承担着数据库的维护工作。

数据库管理系统的主要功能包括以下几个方面。

1. 数据库定义功能

DBMS 一般提供数据定义语言(DDL)，分别定义外模式、模式和内模式。各种模式翻译程序把各种源模式翻译为相应的内部表示，分别称为目标外模式、目标模式和目标内模式。这些目标模式是对数据库的定义，而不是数据本身。它们刻画了数据库的框架，是 DBMS 存取和管理数据的基本依据。

2. 数据存取功能

DBMS 提供数据操纵语言(DML)，从而实现对数据库数据的基本操作：检索、插入、修改和删除。DBMS 控制并执行 DML 语言，完成对数据库的操作。

3. 数据库运行管理

这是 DBMS 运行时的核心部分，包括安全性检查、完整性约束条件的检查和执行、数据库内部的维护等。所有数据库的操作都要在这些控制程序的统一管理下进行，以保证事务的正确运行和数据库的正确有效。

4. 数据库的建立和维护功能

主要包括数据库初始数据的载入和转换、数据库的存储和恢复、数据库的重组、性能监视和分析等功能。

9.1.3 数据库语言

数据库语言一般可分为以下两种：一种是交互式命令语言，它具有语法简明、可独立使用等特点；另一种则嵌入到某种程序设计语言中(如 C、FORTRAN、PASCAL、COBOL 等)，称为宿主型语言。

9.1.4 数据库系统

数据库系统(DataBase System，缩写为 DBS)通常是指带有数据库的计算机应用系统，它不仅包括数据库本身(即实际存储在计算机中的数据)，还包括相应的硬件、软件等。

9.1.5 主要数据模型

数据库领域中过去和现在最常见的数据模型有三种，它们分别是：层次模型(Hierachical Model)、网状模型(Network Model)和关系模型(Relational Model)。其中层次模型和网状模型统称为非关系模型，在关系模型出现以前，它们是常用的数据模型。

关系模型是数据库领域所讨论的模型中最重要的模型。自 20 世纪 80 年代以来，计算机厂商所推出的数据库管理系统的产品几乎都支持关系模型。在用户看来，关系模型中数据库的逻辑结构(即数据结构)就是一张二维表。用二维表结构来表示实体及实体间联系的模型就称为关系模型。

关系模型与以往的非关系模型最大的区别在于，它是建立在严格的数学概念基础之上的，其概念简单清晰，数据库语言易懂易学，用户无需了解复杂的存取路径细节，不需要说明“怎么做”，只需指出“做什么”，就能操作数据库，因此深得用户的青睐和喜爱，并涌现出许多性能良好的商业化关系数据库管理系统(RDBMS)，如著名的 Oracle、Informix、Sybase、MySQL 等。而且关系数据库产品也从单一的集中式系统发展到可在网络环境下运行的分布式系统，从封闭式系统逐步发展到开放式系统，从联机事务处理到支持信息管理、辅助决策，系统的功能不断完善，数据库的应用领域迅速扩大。

9.2 SQL 语言简介

SQL 是英文(Structured Query Language)的缩写，意思为结构化查询语言。SQL 语言的主要功能就是同各种数据库建立联系，进行沟通。按照 ANSI(美国国家标准协会)的规定，SQL 被作为关系型数据库管理系统的标准语言。SQL 语句可以用来执行各种各样的操作，例如更新数据库中的数据，从数据库中提取数据等。目前，绝大多数流行的关系型数据库管理系统(如 Oracle,Sybase, MySQL 等)采用了 SQL 语言标准。虽然很多数据库都对 SQL 语句进行了再开发和扩展，但是包括 Select, Insert, Update, Delete, Create 以及 Drop 在内的标准的 SQL 命令仍然可以被用来完成几乎所有的数据库操作。下面，我们就来详细介绍一下 SQL 语言的基本知识。

9.2.1 数据库表格

一个典型的关系型数据库通常由一个或多个被称作表格的对象组成。数据库中的所有数据或信息都被保存在这些数据库表格中。数据库中的每一个表格都具有自己唯一的表格名称，都是由行和列组成，其中每一列包括了该列名称，数据类型，以及列的其他属性等信息，而行则具体包含某一列的记录或数据。表 9-1 是一个名为学生成绩的数据库表格的实例。

表 9-1 学生成绩的数据库表格

姓 名	学 号	语 文 成 绩	数 学 成 绩
张三	048071	82	85
李军	048072	95	90
张明	048073	93	92
王东	048074	89	88
刘红	048075	92	85

该表格中“姓名”、“学号”、“语文成绩”和“数学成绩”就是 4 个不同的列，而表格中的每一行则包含了具体的表格数据。

9.2.2 数据查询

在众多的 SQL 命令中，select 语句应该算是使用最频繁的。Select 语句主要被用来对数据库进行查询并返回符合用户查询标准的结果数据。Select 语句的语法格式如下：

```
select 列 1 [,列,…] from 表名 [where 条件];
```

[]表示可选项。

select 语句中位于 select 关键词之后的列名用来决定哪些列将作为查询结果返回。用户可以按照自己的需要选择任意列，还可以使用通配符“*”来设定返回表格中的所有列。位于 from 关键词之后的表格名称用来决定将要进行查询操作的目标表格。where 可选从句用来规定哪些数据值或哪些行将被作为查询结果返回或显示。在 where 条件从句中可以使用以下一些运算符来设定查询标准，如表 9-2 所示。

表 9-2 where 条件中的运算符

运 算 符	含 义
=	等于
>	大于
<	小于
>=	大于等于
<=	小于等于
<>	不等于

select 语句举例如下：

```
select 语文成绩, 数学成绩 from 学生成绩 where 学号>048073
```

上面 select 语句的作用是从学生成绩表中，查询学号>048073 的语文成绩和数学成绩。

9.2.3 创建表格

SQL 语言中的 create table 语句被用来建立新的数据库表格。Create table 语句的使用格式如下：

```
create table tablename (column1 data type [constraint], column2 data type [constraint], column3 data type
    [constraint]);
```

简单来说，创建新表格时，在关键词 create table 后面加入所要建立的表格的名称，然后在括号内顺次设定各列的名称、数据类型，以及可选的限制条件等。注意，所有的 SQL 语句在结尾处都要使用“；”符号。

使用 SQL 语句创建的数据库表格和表格中列的名称必须以字母开头，后面可以使用字母，数字或下划线，名称的长度不能超过 30 个字符。注意，用户在选择表格名称时不要使用 SQL 语言中的保留关键词，如 select，create，insert 等作为表格或列的名称。

数据类型用来设定某一个具体列中数据的类型。例如，在姓名列中只能采用 varchar 或 char 的数据类型，而不能使用 number 的数据类型。 SQL 语言中较为常用的数据类型如

表 9-3 所示。

表 9-3 SQL 语言中常用的数据类型

数 据 类 型	说 明
char(size)	固定长度字符串，其中括号中的 size 用来设定字符串的最大长度。Char 类型的最大长度为 255 字节
varchar(size)	可变长度字符串，最大长度由 size 设定
number(size)	数字类型，其中数字的最大位数由 size 设定
number(size,d)	数字类型，size 决定该数字总的最大位数，而 d 则用于设定该数字在小数点后的位数
date	日期类型

最后，在创建新表格时需要注意的一点就是表格中列的限制条件。所谓限制条件就是当向特定列输入数据时所必须遵守的规则。例如，unique 这一限制条件要求某一列中不能存在两个值相同的记录，所有记录的值都必须是唯一的。除 unique 之外，较为常用的列的限制条件还包括 not null 和 primary key 等。Not null 用来规定表格中某一列的值不能为空。Primary key 则为表格中的所有记录规定了唯一的标识符。

create table 语句的用法举例如下：

```
create table student (firstname varchar(15), lastname varchar(20),age number(3), address varchar(30), city
    varchar(20));
```

上面 create table 语句的作用是创建名为 student 的表格，表格中各列的名称分别为 firstname、lastname、age、address、city。它们的数据类型分别是 varchar、varchar、number、varchar、varchar。它们的最大位数分别为 15、20、3、30、20。

9.2.4 向表格中插入数据

SQL 语言使用 insert 语句向数据库表格中插入或添加新的数据行。Insert 语句的使用格式如下：

```
insert into 表名 (第 1 列,...最后一列) values (第 1 个值,...最后一个值);
```

当向数据库表格中添加新记录时，在关键词 insert into 后面输入所要添加的表格名称，然后在括号中列出将要添加新值的列的名称。最后，在关键词 values 的后面按照前面输入的列的顺序对应地输入所有要添加的记录值。例如：

```
insert into employee (firstname, lastname, age, address, city) values (‘Li’, ‘Ming’, 45, ‘No.77 Changan
    Road’, ‘Beijing”);
```

以上 insert 语句的作用是向 employee 表格中插入一条 firstname='Li',

lastname='Ming',age='45',addrss='No.7 Changan Road',city='Beijing'的新记录。

9.2.5 更新记录

SQL 语言使用 update 语句更新或修改满足规定条件的现有记录。Update 语句的格式为：

```
update tablename    set columnname = newvalue [, nextcolumn = newvalue2...] where columnname
    OPERATOR value [and|or column OPERATOR value];
```

使用 update 语句时，关键一点就是要设定好用于进行判断的 where 条件从句。例如：

```
update employee    set age = age+1    where firstname= 'Mary'and lastname= 'Williams';
```

上面语句的作用是把 firstname= 'Mary'且 lastname= 'Williams'的记录中的 age 更新为 age+1。

9.2.6 删除记录

SQL 语言使用 delete 语句删除数据库表格中的行或记录。delete 语句的格式为：

```
delete from tablename    where columnname OPERATOR value [and|or column OPERATOR value];
```

当需要删除某一行或某个记录时，在 delete from 关键词之后输入表格名称，然后在 where 从句中设定删除记录的判断条件。注意，如果用户在使用 delete 语句时不设定 where 从句，则表格中的所有记录将全部被删除。例如：

```
delete from employee    where lastname = May;
```

以上 delete 语句的作用是删除 lastname='May'对应的所有记录。

9.2.7 删除数据库表格

在 SQL 语言中使用 drop table 命令删除某个表格以及该表格中的所有记录。drop table 命令的使用格式为：

```
drop table tablename;
```

如果用户希望将某个数据库表格完全删除，只需要在 drop table 命令后输入希望删除的表格名称即可。drop table 命令的作用与删除表格中的所有记录不同。删除表格中的全部记录之后，该表格仍然存在，而且表格中列的信息不会改变。而使用 drop table 命令则会将整个数据库表格的所有信息全部删除。例如：

```
drop table employee;
```

上述 drop table 语句将完全删除 employee 数据库表格。

以上，我们对 SQL 语言主要的命令和语句进行了较为详细的介绍。在后面的编程中还会用到。

9.3 MySQL 数据库

MySQL 是一个精巧的 SQL 数据库管理系统，虽然它不是开放源代码的产品，但在某些情况下可以自由使用。由于具有功能强大、灵活性好、提供丰富的应用编程接口(ApI)的特点以及精巧的系统结构，MySQL 受到了广大自由软件爱好者甚至商业软件用户的青睐，特别是其与 Apache 和 PHP/PERL 的组合，为建立基于数据库的动态网站提供了强大动力。

9.3.1 MySQL 的安装

安装 MySQL 非常简单。对很多的平台来说，如果你的 Linux 安装没有复制的话，MySQL 网站(http://www.mysql.com)拥有源代码以及预先编译好的二进制文件供下载。通常来说尽管可以使用源代码安装，但是最好找到一个预先编译好的安装。

在 Ubuntu 下安装 MySQL，可以在终端提示符后运行下列命令：

```
$ sudo apt-get install mysql-server mysql-client
```

一旦安装完成，MySQL 服务器应该自动启动。可以在终端提示符后运行以下命令来检查 MySQL 服务器是否正在运行：

```
$ sudo netstat -tap | grep mysql
```

当运行该命令时，可以看到类似下面的行：

```
tcp        0        0 localhost:mysql        *:*                LISTEN
4598/mysqld
```

如果服务器不能正常运行，可以通过下列命令启动它：

```
$ sudo /etc/init.d/mysql restart
```

默认的 MySQL 安装之后根用户(系统管理员 root)是没有密码的，在第一次用根用户登录时，MySQL 会强制你为根用户设定一个密码，在终端提示符后输入下列命令：

```
$ mysql -u root -p
```

这时系统会提示：

```
Enter password:
```

这时输入的密码即为根用户的密码。然后就可以进入到 mysql 中：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 5.0.45-Debian_1ubuntu3.1-log Debian etch distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

在 mysql 提示符后输入 quit 或\q 即可推出 mysql。

```
mysql> quit
Bye
```

在设定了根用户的密码后，再次以根用户登录时，除非提供用户和口令，否则没法运行 mysql，正确的命令如下：

```
$ mysql -u root -ppassword mysql
```

注意，这里的语法有一点特殊，-p 和口令之间没有空格。最后一个参数 mysql 是所选的数据库。如果不提供口令，只输入-p，mysql 将会提示请给出一个口令。由于在命令行中输入口令并不十分妥当(别人可以用 ps 指令等方法来得到这个口令)，所以更好的方法是省略口令，使用这个格式：

```
$ mysql -u root -p mysql
```

然后 mysql 将会提示请提供口令。一旦运行了 mysql，就可以检验一下当前的测试数据库，方法是在 mysql 提示符下输入：

```
mysql> select host,db,user from db;
```

然后将会得到下面形式的列表：

```
+------+---------+------+
| host | db      | user |
+------+---------+------+
| %    | test    |      |
| %    | test\_% |      |
+------+---------+------+
2 rows in set (0.00 sec)
```

9.3.2　MySQL 管理

MySQL 自带了少量实用程序，以便于管理这个系统。在编写 MysQL 客户程序以前，需简要地浏览一下这些实用程序。

9.3.2.1　命令

所有命令都有三个标准参数：-u Username –p [Password] –h host。

-u 参数用于输入登录 mysql 的用户，-h 参数用于在不同的主机上连接一台服务器，对于本地服务器则此参数可以省略。如果给出-p 却不提供口令的话，则会提示输入口令。如果没有-p 参数，则 MySQL 命令假设为不需要口令。

1. mysql

这是标准命令行工具，可以用于以后将要涉及到的很多管理和权限任务。

mysql 命令需要一个附加的参数，就是在选项后面加上需要连接的数据库名称。例如，对于密码为 bar 的用户 rick，为了用 mysql 开启选定的数据库 foo，需要输入：

```
$ mysql -u rick -pbar foo
```

通常用这种方法可以很容易地指定要连接的数据库。可以通过用-h 选项打开 mysql 的方法来显示其他的选项。

2. mysqladmin

mysqladmin 是主要的管理实用程序。除了通用的-u user 以及-p 来提示输入口令以外，还有几个主要的命令选项，如表 9-4 所示。

表 9-4　mysqladmin 其他命令选项

命　　令	说　　明
create dbname	创建一个名为 dbname 的新数据库
drop dbname	删除 dbname 数据库
flush-tables	清洗所有的表
password newpassword	用 newpassword 变更原有口令
shutdown	关掉 MySQL 服务器
status	给出服务器的简短状态信息
variables	打印出所有可用变量
version	给出服务器的版本信息

例如：

```
$ mysqladmin -u root -p status
```

执行上述命令，输入口令后，系统显示如下：

```
Uptime: 4635   Threads: 1   Questions: 86   Slow queries: 0   Opens: 23   Flush tables: 1   Open tables:
17   Queries per second avg: 0.019
```

若要查看服务器版本信息，可以输入如下命令：

```
$ mysqladmin -u root -p version
```

输入口令后，系统显示：

```
mysqladmin   Ver 8.41 Distrib 5.0.45, for pc-linux-gnu on i486
Copyright (C) 2000-2006 MySQL AB
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL license

Server version          5.0.45-Debian_1ubuntu3.1-log
Protocol version        10
Connection              Localhost via UNIX socket
UNIX socket             /var/run/mysqld/mysqld.sock
Uptime:                 1 hour 18 min 31 sec

Threads: 1   Questions: 87   Slow queries: 0   Opens: 23   Flush tables: 1   Open tables: 17   Queries
per second avg: 0.018
```

更改根用户的口令：

```
$ mysqladmin -u root -p password pass123
```

在系统提示输入原有口令后，root 用户的口令将更改为 pass123。

如果只以-u Username 的方式打开程序，mysqladmin 会提供一个详细的命令列表，读者可以参考。

3. mysqldump

mysqldump 可以将一个数据库(所有的表或选定的表)导出到一个文件中，它的基本用法是：

```
mysqldump [OPTIONS] database [tables]
```

除了基本的-u 和-p 选项外，常用的其他选项如表 9-5 所示。

表 9-5　mysqldump 其他命令选项

命　　令	说　　明
--add-locks	在每个表导出之前锁定表并且之后解锁表
--add-drop-table	在每个 create 语句之前增加一个 drop table
--allow-keywords	允许创建包含关键字的列
-c, --complete-insert	使用完整的 insert 语句

(续表)

命　　令	说　　明
-C, --compress	如果客户和服务器均支持压缩，压缩2者间的所有信息
--delayed	用 INSERT DELAYED 命令插入行
-d, --no-data	不写入表的任何行信息，只导出表结构
-t, --no-create-info	不写入表创建信息(CREATE TABLE 语句)
--help	显示帮助信息

mysqldump 的输出信息在终端上显示，可将它重新定位到文件中。

使用这个实用程序可以对 mysql 数据库进行定期的备份，或者输出数据以便移到另一个数据库中。输出的格式是直接 ASCII 格式，非常易于阅读；不仅如此，输出的内容中还结合有注释内容。例如，将数据库 testdb 导出到文件 testdb.bak 中，用下面的命令：

```
$ mysqldump -u root -p testdb>testdb.bak
```

导出成功后，testdb.bak 的内容如下所示：

```
-- MySQL dump 10.11
--
-- Host: localhost    Database: testdb
-- ------------------------------------------------------
-- Server version        5.0.45-Debian_1ubuntu3.1-log

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE,
SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `children`
--

DROP TABLE IF EXISTS `children`;
CREATE TABLE `children` (
```

```
  `childno` int(11) NOT NULL auto_increment,
  `fname` varchar(30) default NULL,
  `age` int(11) default NULL,
  PRIMARY KEY  (`childno`)
) ENGINE=MyISAM AUTO_INCREMENT=3 DEFAULT CHARSET=latin1;

--
-- Dumping data for table `children`
--

LOCK TABLES `children` WRITE;
/*!40000 ALTER TABLE `children` DISABLE KEYS */;
INSERT INTO `children` VALUES (1,'Jenny',14),(2,'Jack',12);
/*!40000 ALTER TABLE `children` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2008-05-24  3:38:45
```

4. mysqlimport

这个实用程序是伴随着mysqldump一起的。它可以使数据库从文本文件中被重新创建。通常这些文本文件是由mysqldump创建的，通常所需要的参数仅仅是数据库名称以及要读取命令的文本文件名称。

5. mysqlshow

这是一个非常便利的实用小程序，它根据所设定的参数显示服务器、数据库或者表的信息。

- 没有参数时，它将列出所有可用的数据库。
- 参数是一个数据库时，它将列出这个数据库中所有的表。
- 参数是一个数据库以及一个表名称时，它将列出这个表中的列。
- 参数是数据库、表和列时，它将列出指定列的信息。

通常提供列名称没有太大的意义，因为在表级别上可以显示每一列的信息。

如显示所有的数据库，命令如下：

```
$ mysqlshow -u root -p
```

执行后，系统显示如下：

```
+--------------------------+
|          Databases        |
+-------------------------- +
| information_schema    |
| mysql                       |
| testdb                      |
+--------------------------+
```

可以看出系统中共有 3 个数据库，分别是：information_schema、mysql、testdb。

显示数据库 testdb 中所有的表：

```
$ mysqlshow -u root –p testdb
```

系统显示如下：

```
Database: testdb
+----------+
|    Tables |
+----------+
| children |
+----------+
```

可以看出，testdb 中只有一个表 children。

显示 testdb 中 children 表中的所有列：

```
$ mysqlshow -u root -p testdb children
```

系统显示如图 9-1 所示。

```
Database: testdb  Table: children
+---------+------------+-----------------+-----+----+--------+--------------+------------------------------+--------+
| Field   | Type       | Collation       | Null| Key| Default| Extra        | Privileges                   | Comment|
+---------+------------+-----------------+-----+----+--------+--------------+------------------------------+--------+
| childno | int(11)    |                 | NO  | PRI|        | auto_increment| select,insert,update,references |     |
| fname   | varchar(30)| latin1_swedish_ci| YES |    |        |              | select,insert,update,references |     |
| age     | int(11)    |                 | YES |    |        |              | select,insert,update,references |     |
+---------+------------+-----------------+-----+----+--------+--------------+------------------------------+--------+
```

图 9-1　显示 testdb 中 children 表中的所有列

9.3.2.2　创建用户并提供权限

除了备份重要数据外，最常进行的管理工作就是设置用户权限了。从 MySQL 的 3.22 版本开始，用户权限通过两个 SQL 命令来管理：grant 和 revoke。这两个指令实质是通过操作 user(连接权限和全局权限)、db(数据库级权限)、tables_priv(数据表级权限)、columns_priv(数据列级权限)四个权限表来分配权限的。host 权限表不受这两个指令影响。

下面将会详细介绍用户权限管理的内容。这两个命令都在 mysql 命令程序中运行。

1. grant

MySQL 的 grant 命令类似于 SQL92 的标准，但仍然有显著的区别，通常的格式为：

```
GRANT privileges (columns) ON what TO account IDENTIFIED BY 'password' REQUIRE encryption
requirements WITH grant or resource management options;
```

其中，privileges 表示授予的权限，mysql 中的权限如表 9-6 所示。columns 表示作用的列(可选)，what 设置权限级别可以是全局级、数据库级、数据表级和数据列级，account 是权限授予的用户，用"user_name"@"host_name"这种用户名、主机名格式，IDENTIFIED BY 'password'设置用户账号口令，REQUIRE encryption requirements 设置经由 SSL 连接账号，WITH grant or resource management options 设置账号的管理和资源(连接服务器次数或查询次数等)选项。

表 9-6　mysql 中的权限

权　　限	说　　明
CREATE TEMPORARY TABLE	创建临时数据表
FILE	操作系统文件
GRANT OPTION	可把本账号的权限授予其他用户
LOCK TABLES	锁定指定数据表
PROCESS	查看运行着的线程信息
RELOAD	重新加载权限表或刷新日志及缓冲区
REPLICATION CLIENT	可查询主/从服务器主机名
REPLICATION SLAVE	运行一个镜像从服务器
SHOW DATABASES	可运行 SHOW DATABASES 指令
SHUTDOWN	关闭数据库服务器
SUPER	可用 kill 终止线程以及进行超级用户操作
ALTER	可修改表和索引的结构
CREATE	创建数据库和数据表
DELETE	删除数据表中的数据行
DROP	删除数据表和数据行
INDEX	建立或删除索引
INSERT	插入数据行
SELECT	查询数据行
UPDATE	更新数据行
ALL	所有权限，但不包括 GRANT
USAGE	无权限

由 ON 子句设置的权限作用范围如表 9-7 所示。

表 9-7　mysql 中的权限作用范围

权限限定符	说　　明
.	全局级权限，作用于所有数据库
*	全局级权限，若未指定默认数据库，其作用范围是所有数据库，否则，其作用范围是当前数据库
dbname.*	数据库级权限，作用于指定数据库里的所有数据表
dbname.tablename	数据表级权限，作用于数据表里的所有数据列
tablename	数据表级权限，作用于默认数据库中指定的数据表里的所有数据列

例如：

```
mysql> grant all on db.* to'test'@'localhost' identified by 'test';
```

运行上述命令后，test 用户只能通过'test'密码从本机访问 db 数据库

```
mysql> grant all on db.* to'test'%'localhost' identified by 'test';
```

运行上述命令后，test 用户可通过‘test’密码从任意计算机上访问 db 数据库。其中，'%'代表任意字符，'_'代表一个任意字符。主机名部分还可以是 IP 地址。如果没有给定主机部分，则默认为任意主机，也就是'test'和'test'@'%'是等价的。

USAGE 权限通常用来修改与权限无关的账户项，如：

```
mysql>GRANT USAGE ON *.* TO account IDENTIFIED BY 'new_password';
```

上述命令的作用是修改密码。

```
mysql>GRANT USAGE ON *.* TO account REQUIRE SSL;
```

上述命令启用 SSL 连接。

拥有 WITH GRANT OPTION 权限的用户可把自己所拥有的权限转授给其他用户，如：

```
mysql>GRANT ALL ON db.* TO 'test'@'%' IDENTIFIED BY 'password' WITH GRANT OPTION;
```

这样 test 用户就有权把该权限授予其他用户。

GRANT 也可以限制资源使用，如：

```
mysql>GRANT ALL ON db.* TO account IDENTIFIED BY 'password' WITH
MAX_CONNECTIONS_PER_HOUR 10 MAX_QUERIES_PER_HOUR 200
MAX_UPDATES_PER_HOUR 50;
```

允许 account 用户每小时最多连接 20 次服务器，每小时最多发出 200 条查询命令(其中更新命令最多为 50 条)。

默认都是零值，即没有限制。FLUSH USER_RESOURCES 和 FLUSH PRIVILEGES 可对资源限制计数器清零。

2. revoke 和 delete

revoke 用来取消权限，它的用法如下：

```
mysql>REVOKE privileges (columns) ON what FROM account;
```

各个选项的含义与 grant 中的相同。

例如：

```
mysql>REVOKE SELECT ON db.* FROM 'test'@'localhost';
```

执行上述命令后，将删除 test 账号从本机查询 db 数据库的权限。

REVOKE 可删除权限，但不能删除账号，即使账号已没有任何权限。所以 user 数据表里还会有该账号的记录，要彻底删除账号，需用 DELETE 命令删除 user 数据表的记录，如：

```
mysql>DELETE FROM user where User='test' and Host='localhost';
mysql>fulsh privileges;
```

REVOKE 不能删除 REQUIRE 和资源占用的配置。他们是要用 GRANT 来删除的，如：

```
GRANT USAGE ON *.* TO account REQUIRE NONE;
GRANT USAGE ON *.* TO account WITH MAX_CONNECTIONS_PER_HOUR 0
MAX_QUERIES_PER_HOUR 0 MAX_UPDATES_PER_HOUR 0;
```

上述命令删除 account 账号的 SSL 连接选项和 account 账号的资源限制。

9.3.2.3 口令

在创建新用户时，如果忘记指定口令，可以在后面对其进行设定。首先以根用户登录，并选定 mysql 数据库。输入下列查询语句：

```
mysql> select host,user,password from user;
```

则会得到类似于图 9-2 所示的一张表。

```
+------------+----------------+-------------------------------------------+
| host       | user           | password                                  |
+------------+----------------+-------------------------------------------+
| localhost  | root           | *33D4879D624B8B3F516EF6BCC9D0FCE9516DF65B |
| lxy-desktop| root           | *33D4879D624B8B3F516EF6BCC9D0FCE9516DF65B |
| 127.0.0.1  | root           | *33D4879D624B8B3F516EF6BCC9D0FCE9516DF65B |
| localhost  | debian-sys-maint| *EA6DA628CFB1CA08FD0E0D2B4A406B63AB1CC4A7 |
| localhost  | test           | *94BDCEBE19083CE2A1F959FD02F964C7AF4CFC29 |
| %          | test           | *94BDCEBE19083CE2A1F959FD02F964C7AF4CFC29 |
| localhost  | test1          |                                           |
+------------+----------------+-------------------------------------------+
7 rows in set (0.00 sec)
```

图 9-2　查询 user 表

如果要为用户 test1 分配口令 test 的话，可以输入下列命令：

```
mysql> update user set password=password('test') where user ='test1';
```

再次显示 mysql.user 表中的相关列：

```
mysql> select host,user,password from user;
```

执行结果如图 9-3 所示。

```
+-------------+------------------+-------------------------------------------+
| host        | user             | password                                  |
+-------------+------------------+-------------------------------------------+
| localhost   | root             | *33D4879D624B8B3F516EF6BCC9D0FCE9516DF65B |
| lxy-desktop | root             | *33D4879D624B8B3F516EF6BCC9D0FCE9516DF65B |
| 127.0.0.1   | root             | *33D4879D624B8B3F516EF6BCC9D0FCE9516DF65B |
| localhost   | debian-sys-maint | *EA6DA628CFB1CA08FD0E0D2B4A406B63AB1CC4A7 |
| localhost   | test             | *94BDCEBE19083CE2A1F959FD02F964C7AF4CFC29 |
| %           | test             | *94BDCEBE19083CE2A1F959FD02F964C7AF4CFC29 |
| localhost   | test1            | *94BDCEBE19083CE2A1F959FD02F964C7AF4CFC29 |
+-------------+------------------+-------------------------------------------+
7 rows in set (0.01 sec)
```

图 9-3　user 表的内容

可以看到，test1 与我们在前面为用户 test 定义的加密口令是一样的。

9.3.2.4　创建数据库

创建数据库的命令如下：

```
create database dbname;
```

其中，dbname 是欲创建的数据库的名字。

例如，创建一个 testdb 的数据库：

```
mysql> show databases;
+----------------------+
| Database             |
+----------------------+
| information_schema |
| mysql                |
+-----------------------+
2 rows in set (0.01 sec)

mysql> create database testdb;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+-----------------------+
```

```
| Database               |
+------------------------+
| information_schema |
| mysql                  |
| testdb                 |
+------------------------+
3 rows in set (0.00 sec)
```

其中，show databases;命令是查看系统中当前存在的数据库，可以看出在执行 create 命令后，成功地创建了 testdb 数据库。然后切换使用 use 命令使用数据库 testdb：

```
mysql> use testdb
Database changed
```

现在我们可以按照需要创建任何表。首先看一下现在数据库中存在的表：

```
mysql> show tables;
Empty set (0.00 sec)
```

说明刚才建立的数据库中还没有数据库表。下面来创建一个数据库表 children，该表显示一个学校的儿童登记表，表的内容包含 ID、儿童姓名、年龄。命令如下：

```
mysql>   create table children(childno INTEGER AUTO_INCREMENT NOT NULL PRIMARY
         KEY,fname VARCHAR(30),age INTEGER);
Query OK, 0 rows affected (0.09 sec)
```

由于 fname 的列值是变化的，因此选择 VARCHAR，其长度不一定是 30。可以选择从 1 到 255 的任何长度，如果以后需要改变它的字长，可以使用 ALTER TABLE 语句。

创建了一个表后，看看刚才做的结果，用 SHOW TABLES 显示数据库中有哪些表：

```
mysql> show tables;
+--------------------+
| Tables_in_testdb |
+--------------------+
| children           |
+--------------------+
1 row in set (0.00 sec)
```

可以用 DESCRIBE 查看表的结构，命令及显示结果如图 9-4 所示。

由于 children 为新建的表，还没有任何记录。这可以通过下列查询命令看出：

```
mysql> select * from children;
Empty set (0.09 sec)
```

```
mysql> DESCRIBE children;
+---------+-------------+------+-----+---------+----------------+
| Field   | Type        | Null | Key | Default | Extra          |
+---------+-------------+------+-----+---------+----------------+
| childno | int(11)     | NO   | PRI | NULL    | auto_increment |
| fname   | varchar(30) | YES  |     | NULL    |                |
| age     | int(11)     | YES  |     | NULL    |                |
+---------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

图 9-4　显示表的结构

说明刚才创建的表还没有记录。 加入 3 条新记录，命令如下：

```
mysql> insert into children(fname,age) values("Jenny",14);
Query OK, 1 row affected (0.05 sec)

mysql> insert into children(fname,age) values("John",10);
Query OK, 1 row affected (0.00 sec)

mysql> insert into children(fname,age) values("Jack",11);
Query OK, 1 row affected (0.00 sec)
```

然后再次运行 select 命令，结果如图 9-5 所示。

```
mysql> select * from children;
+---------+-------+------+
| childno | fname | age  |
+---------+-------+------+
|       1 | Jenny |   14 |
|       2 | John  |   10 |
|       3 | Jack  |   11 |
+---------+-------+------+
3 rows in set (0.01 sec)
```

图 9-5　select 查询结果

可以按照上面的方法一条一条地将所有儿童的记录加入到表中。

以上我们简要介绍了 MySQL 数据库的安装和一些常用的使用方法，关于 MySQL 更详细的介绍，读者可以查阅 MySQL 的帮助手册。下一节我们将介绍用 C 语言访问 MySQL 数据库的方法。

9.4　用 C 语言访问 MySQL 数据库

MySQL 可以用很多不同的语言进行访问，其中包括：C、C++、Java、Perl、Python、PHP 等。在这一节，我们介绍用 C 语言访问 MySQL 数据库的方法。

9.4.1 连接数据库

安装了 MySQL 后，在/usr/include/mysql 下，包含了用 C 语言操作 MySQL 所需的头文件：mysql.h。其中定义了相关的数据结构，它们的意义如表 9-8 所示。

表 9-8 mysql.h 中定义的结构

结 构	说 明
MYSQL	连接句柄
MYSQL_RES	用来保存从数据库中检索出的列
MYSQL_ROW	保存返回的其中一列
MYSQL_FIELD	数据库中的字段

用 C 语言向一个 MySQL 数据库的连接包括 2 步：

(1) 初始化一个 MYSQL 结构。

(2) 进行连接。

首先必须调用 mysql_init 连接初始化一个 MYSQL 结构，它的原型如下：

```
#include <mysql/mysql.h>
MYSQL *mysq_init(MYSQL *mysql);
```

其中，如果传递的参数为 NULL，则初始化一个新的结构，分配新的内存空间，其他情况下将把 mysql 初始化。

函数成功返回初始化后的 MYSQL 结构指针，否则返回 NULL。

在初始化 MYSQL 结构之后，接下来需要设置连接数据库的相关参数，可以用 mysql_real_connect 函数来设置这些参数，并进行实际的连接，它的原型如下：

```
#include <mysql/mysql.h>
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd,
   const char *db, unsigned int port, const char *unix_socket, unsigned int client_flag);
```

其中，mysql 指向刚才用 mysql_init 初始化的结构体。host 是 MySQL 服务器所在的服务器计算机的名称或者 IP 地址。如果你要连接本地计算机，那么应该使用 localhost，而不是计算机名，这样 MySQL 可以优化连接类型。

user 以及 passwd 是数据库登录时的用户名和口令。如果登录的用户名是 NULL，则登录的用户名假定为当前的登录 ID。如果口令是 NULL，则你只可以访问这个服务器中不需要口令的数据。口令在传递到网络以前是加密的。

除非需要非标准值，否则端口号以及 unix_socket 的值分别为 0 和 NULL。这两个值是默认的。

client_flag 值通常是 0，但是在很特殊的情况下可以被设置为下列标志的组合，如表

9-9 所示。

表 9-9 client_flag 选项

选　　项	说　　明
CLIENT_FOUND_ROWS	返回找到的(匹配的)行数，不是受到影响的行数
CLIENT_NO_SCHEMA	不允许 db_name.tbl_name.col_name 语法，如果使用该语法，会导致语法分析器产生一个错误，它是为一些 ODBC 程序捕捉错误所使用的
CLIENT_COMPRESS	使用压缩协议
CLIENT_ODBC	客户是一个 ODBC 客户。这使 mysqld 变得对 ODBC 更友好

mysql_real_connect 函数成功返回一个指向 MYSQL 结构的指针，返回值与第一个参数值相同。如果连接失败，返回 NULL。可以用 mysql_error 函数查看错误原因，下面会介绍这一函数。

在完成连接任务(通常在程序结束以后)，可以用 mysql_close 来关闭连接，它的原型如下：

```
#include <mysql/mysql.h>
void mysql_close(MYSQL *mysql);
```

这样就可以关闭连接。参数 mysql 将被清空，指针也变为无效而不能重新使用。

与连接函数紧密关联(它只能在 mysql_init 和 mysql_real_connect 之间调用)的是 mysql_options，原型如下：

```
#include <mysql/mysql.h>
int mysql_options(MYSQL *mysql, enum mysql_option option, const char *arg);
```

mysql_options 用于设置额外的连接选项，并影响连接的行为。可多次调用该函数来设置数个选项。

应在 mysql_init()之后、以及 mysql_real_connect()之前调用 mysql_options。

选项参量指的是打算设置的选项。Arg 参量是选项的值。如果选项是整数，那么 arg 应指向整数的值。常用的选项值如表 9-10 所示。

表 9-10 arg 选 项

选　　项	说　　明
MYSQL_INIT_COMMAND	连接到 MySQL 服务器时将执行的命令。再次连接时将自动地再次执行
MYSQL_OPT_CONNECT_TIMEOUT	以秒为单位的连接超时
MYSQL_OPT_PROTOCOL	要使用的协议类型。应是 mysql.h 中定义的 mysql_protocol_type 的枚举值之一
MYSQL_OPT_COMPRESS	在网络连接中使用压缩

mysql_options 成功调用返回 0，失败返回非 0 值。

例 9-1 下面的代码段将连接超时设定为 7 秒。

```
unsigned int timeout=7;
...
MYSQL *connection=mysql_init(NULL);
ret=mysql_options(connection, MYSQL_OPT_CONNECT_TIMEOUT, (const char *)&timeout);

if(ret)
{
    /*错误处理*/
    ...
}
connection=mysql_real_connect(connection,...);
```

以上我们已经知道了怎样对一个连接进行基本的设定以及怎样关闭它。现在写一个短程序来验证一下。

例 9-2 连接 MYSQL 数据库。

```
1    /*ex1.c*/
2    #include <stdlib.h>
3    #include <stdio.h>
4    #include <mysql/mysql.h>
5    MYSQL *conn_ptr;
6
7    int main()
8    {
9        conn_ptr=mysql_init(NULL);
10       if(!conn_ptr)
11       {
12           fprintf(stderr,"mysql_init failed!\n");
13           return -1;
14       }
15       conn_ptr=mysql_real_connect(conn_ptr,"localhost","test","test","testdb", 0, NULL,0);
16       if(conn_ptr)
17           printf("Connection succeed!\n");
18       else
19       {
20           printf("Connection failed!\n");
21           return -2;
22       }
23       mysql_close(conn_ptr);
24       printf("Connection closed.\n");
```

```
25          return 0;
26      }
```

说明：上面的程序首先初始化一个 MYSQL 结构指针(第 9~14 行)，然后连接本地 MYSQL 服务器，用户名是 test，口令是 test，连接的数据库是 testdb(第 15 行)，随后关闭连接，退出程序(第 23~25 行)。

由于要用到 mysql 库文件，因此在编译时需要指定，编译连接命令如下：

```
gcc -lmysqlclient -o ex1 ex1.c
```

本章后面的例子程序都要加-lmysqlclient 参数，不再另行说明。

程序执行结果如下：

```
$ ./ex1
Connection succeed!
Connection closed.
```

可见，连接一个数据库是非常简单的。

9.4.2　错误处理

在进入更有用的程序之前，我们需要看一下 MYSQL 是怎样处理错误的。所有的错误都通过返回编码来指示，并且通过连接句柄结构体来报告细节。有 2 个函数可以进行错误报告：

```
#include <mysql/mysql.h>
unsigned int mysql_errno(MYSQL *mysql);
const char *mysql_error(MYSQL *mysql);
```

对于由 mysql 指定的连接，mysql_errno()返回最近调用的函数的错误代码，该函数调用可能成功也可能失败。0 返回值表示未出现错误。在 MySQL errmsg.h 头文件中，列出了客户端错误消息编号。

对于由 mysql 指定的连接，对于失败的最近调用的函数，mysql_error 返回包含错误消息的、由 Null 终结的字符串。如果该函数未失败，mysql_error 的返回值可能是以前的错误，或指明无错误的空字符串。

例 9-3　mysql 错误处理函数。

```
1       /*ex2.c*/
2       #include <stdlib.h>
3       #include <stdio.h>
4       #include <mysql/mysql.h>
5
6       int main()
```

```
7    {
8         MYSQL my_connection;
9         mysql_init(&my_connection);
10        if(mysql_real_connect(&my_connection,"localhost","test"," ","testdb", 0, NULL, 0))
11        {
12             printf("Connection Succeed.\n");
13             mysql_close(&my_connection);
14        }
15        else
16        {
17             fprintf(stderr,"Connection failed.\n");
18             if(mysql_errno(&my_connection))
19             {
20                  fprintf(stderr,"Connection error: %d %s\n",mysql_errno(&my_connection),
21                       mysql_error(&my_connection));
22                  return -1;
23             }
24        }
25        return 0;
26   }
```

说明：例 9-3 与例 9-2 基本相同，不同的是在连接时，没用说明口令(第 10 行)，因此 mysql_real_connect 函数会出错，随后用 mysql_errno、mysql_error 函数输出错误代码和错误提示信息(第 18~23 行)。

程序运行结果如下：

```
$ ./ex2
Connection failed.
Connection error: 1045 Access denied for user 'test'@'localhost' (using password: YES)
```

从错误提示信息可以看出，是由于口令不正确导致连接失败。

9.4.3 执行 SQL 语句

我们已经有一个连接，并且知道怎样处理错误，那么现在就可以做一些关于数据库的实际工作了。执行 SQL 语句的函数是 mysql_query：

```
#include <mysql/mysql.h>
int mysql_query(MYSQL *mysql, const char *query);
```

mysql_query 函数将一个指针指向一个连接结构，并且执行包含 SQL 的文本字符串。与命令行工具不同，它没有用来表示终止的分号。

如果运行成功，函数返回 0，失败返回非 0，可能的错误信息如表 9-11 所示。

表 9-10 mysql_query 错误信息

选　　项	说　　明
CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令
CR_SERVER_GONE_ERROR	MySQL 服务器不可用
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失
CR_UNKNOWN_ERROR	出现未知错误

对于 SQL 语句，可以包含以下几种情况：

9.4.3.1　不返回数据的 SQL 语句

这些语句包括 UPDATE、DELETE 以及 INSERT 语句。由于它们不从数据库返回数据，因此更容易学习。

另一个需要介绍的重要函数是用来检验受影响的行数量的函数：

```
#include <mysql/mysql.h>
my_ulonglong mysql_affected_rows(MYSQL *mysql)
```

这个函数返回上次 UPDATE 更改的行数，上次 DELETE 删除的行数或上次 INSERT 语句插入的行数。对于 UPDATE、DELETE 或 INSERT 语句，可在 mysql_query 后立刻调用。出于可移植性的原因，这个结果是没有符号的。在用于 printf 的时候，推荐设定类型为无符号的长数据，用%lu 来指定这个格式。

下面看一个例子。

例 9-4　查看 SQL 语句改动的行数量。

```
1     /*ex3.c*/
2     #include <stdlib.h>
3     #include <stdio.h>
4
5     #include <mysql/mysql.h>
6
7     int main()
8     {
9             MYSQL my_connection;
10            int res;
11            mysql_init(&my_connection);
12            if(mysql_real_connect(&my_connection,"localhost","test","test","testdb", 0, NULL,0))
13            {
14                    printf("Connection Succeed.\n");
15                    res=mysql_query(&my_connection,"INSERT INTO    children(fname,age)
                      VALUES('Ann',3)");
16                    if(!res)
```

```
17                  printf("Inserted %lu    rows\n",(unsigned
                        long )mysql_affected_rows(&my_connection));
18              else
19              {
20                  fprintf(stderr, "Insert error %d %s\n",mysql_errno(&my_connection),
                        mysql_error(&my_connection));
21                  return -1;
22              }
23              mysql_close(&my_connection);
24              printf("Connection closed.\n");
25          }
26          else
27          {
28              fprintf(stderr,"Connection failed.\n" );
29              if(mysql_errno(&my_connection))
30              {
31                  fprintf(stderr,"Connection error: %d
                    %s\n",mysql_errno(&my_connection),mysql_error(&my_connection));
32              return -2;
33              }
34          }
35          return 0;
36      }
```

说明：这个程序与例 9-3 基本相同，在连接成功后，调用了 mysql_query 函数向 testdb 数据库的 children 表中插入了一条新的记录，并输出了插入的行数(第 12~17 行)，其他部分与例 9-3 相同，在此不再赘述。

程序执行结果如下：

```
$ ./ex3
Connection Succeed.
Inserted 1    rows
Connection closed.
```

从执行结果可以看出，插入的行数为 1。

9.4.3.2　返回数据的语句

下面我们要了解 SQL 最常用的功能，即用 SELECT 语句从数据库中检索数据。

通常从 MySQL 数据库中检索数据有 4 个步骤：

(1) 发出查询。

(2) 检索数据。

(3) 处理数据。

(4) 整理所需要的数据。

用 mysql_query 发出查询。检索数据可以用 mysql_store_result 或者 mysql_use_result，这取决于希望以何种方式检索数据，接着是调用 mysql_fetch_row 来处理数据。最后必须调用 mysql_free_result 以允许 MySQL 进行必要的整理工作。

在单一的调用中使用 SELECT(或者其他返回数据的语句)检索数据，方法是使用 mysql_store_result，它的原型如下：

```
#include <mysql/mysql.h>
MYSQL_RES *mysql_store_result(MYSQL *mysql);
```

这个函数必须在 mysql_query 检索完数据以后才能调用，以用来在结果集合中存储数据。此函数从服务器中检索所有的数据并将它立即存储在客户端。它返回一个我们以前没有遇到过的结构体指针——结果集合结构体指针。如果这个语句失败，则返回 NULL 值。

假如没有返回 NULL 值，则可以调用 mysql_num_rows 函数来检索实际返回的行数，这个数当然有可能为 0。mysql_num_rows 函数的原型如下：

```
#include <mysql/mysql.h>
my_ulonglong mysql_num_rows(MYSQL_RES *result);
```

mysql_num_rows 函数在结果集合中返回行的数量。返回在结果集合中的行数有可能为 0。如果 mysql_store_result 成功，则 mysql_num_rows 也将成功。

将 mysql_store_result 和 mysql_num_rows 结合来检索数据是非常简便和直观的方法。一旦 mysql_store_result 返回成功，所有查询的数据都将存储在客户端，这样我们可以在结果结构体中查询这些数据，由于这些数据现在对于我们的程序而言是在当地，所以不需要冒更多数据库或网络出错的危险。这样我们也可以及时发现返回的行数量，也可以使编码变得更简单。像在前面提到过的一样，将立刻将结果返回给客户。不过对于更大的结果集合而言，这种方法会大量地消耗服务器、网络、以及客户资源。因此，如果我们要处理的是更大的数据集合，最好还是按照需要检索数据。

在检索到数据后需要对数据进行处理，可以用以下几个函数实现 mysql_fetch_row、mysql_data_seek、mysql_row_tell、mysql_row_seek。

下面先看一下 mysql_fetch_row 函数，它的原型如下：

```
#include <mysql/mysql.h>
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)
```

这个函数获得从 store result 中得到的结果结构体，并从中检索单个行，返回分配的行结构体中的数据。当没有更多的数据，或者出错时，将返回 NULL 值。我们后面将回到这个行结构体中处理数据。

与 mysql_fetch_row 不同的是，mysql_data_seek 函数在一个查询结果集合中定位任意行。它的原型如下：

```
#include <mysql/mysql.h>
void mysql_data_seek(MYSQL_RES *result, unsigned long long offset);
```

这个函数允许用户进入结果集，设置将由下一个获取操作返回的行。offset 是行号，它必须在从 0 到结果集中的行数减 1 的范围内。传递 0 将导致在下一次调用 mysql_fetch_row 时返回第一行。

mysql_row_tell 函数的原型如下：

```
#include <mysql/mysql.h>
MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result)
```

这个函数返回一个偏移值，它表示结果集中的当前位置。它不是行号，不能将它用于 mysql_data_seek。但是，可将它用于 mysql_row_seek：

```
#include <mysql/mysql.h>
MYSQL_ROW_OFFSET mysql_row_seek(MYSQL_RES *result, MYSQL_ROW_OFFSET offset);
```

mysql_row_seek 移动结果集中的当前位置，并返回以前的位置。

有时，这一对函数对于在结果集中的已知点之间跳转很有用。请注意，不要将 mysql_row_tell 和 mysql_row_seek 使用的偏移值与 mysql_data_seek 使用的行号混淆，这 2 个值是不能互换的。

检索数据最后一个需要了解的函数是 mysql_free_result。它的原型如下：

```
#include <mysql/mysql.h>
void mysql_free_result(MYSQL_RES *result);
```

当已经完成一个结果集合时，则必须调用此函数以便 MySQL 库整理分配的对象。

以上介绍了从 MySQL 数据库中检索数据需要的几个函数，下面看一个例子。

例 9-5 从 MySQL 数据库中检索数据。

```
1     /*ex4.c*/
2      #include <stdlib.h>
3      #include <stdio.h>
4      #include    <mysql/mysql.h>
5      MYSQL my_connection;
6     MYSQL_RES *res_ptr;
7     MYSQL_ROW sqlrow;
8     int main()
9     {
10        int res;
11        mysql_init(&my_connection);
12        if (mysql_real_connect(&my_connection, "localhost", "test","test", "testdb", 0, NULL, 0))
13             {
14              printf("Connection success\n");
```

```
15              res = mysql_query(&my_connection, "SELECT childno, fname,    age FROM
                    children
16                                  WHERE age > 5");
17          if (res)
18                 {
19               printf("SELECT error: %s\n", mysql_error(&my_connection));
20                    return -2;
21                 }
22              else
23                 {
24                   res_ptr = mysql_store_result(&my_connection);
25                   if (res_ptr)
26                   {
27                        printf("Retrieved %lu rows\n",(unsigned long)mysql_num_rows(res_ptr));
28                        while ((sqlrow = mysql_fetch_row(res_ptr)))
29                     printf("Fetched data...\n");
30                        if (mysql_errno(&my_connection))
31                        {
32                          fprintf(stderr, "Retrive error: s\n",mysql_error(&my_connection));
33                          return -3;
34                        }
35               }
36               mysql_free_result(res_ptr);
37            }
38        mysql_close(&my_connection);
39        printf("Connection closed.\n");
40      }
41      else
42      {
43       fprintf(stderr, "Connection failed\n");
44       if (mysql_errno(&my_connection))
45             {
46            fprintf(stderr, "Connection error %d: %s\n",
47              mysql_errno(&my_connection),mysql_error(&my_connection));
48                 return -1;
49          }
50        }
51      return 0;
52   }
```

说明： 在上面的程序中首先初始化一个 MYSQL 结构(第 47 行)，然后建立连接(第 48 行)。在数据库 testdb 的 children 表中选定 age 大于 5 的行的内容(第 51 行)。然后检索返回的结果集并循环通过已检索的数据，最后调用 mysql_free_result 函数使 MySQL 库整理分配

的对象，关闭连接(第 60~74 行)。

程序的执行结果如下：

```
$ ./ex4
Connection success
Retrieved 3 rows
Fetched data...
Fetched data...
Fetched data...
Connection closed.
```

为了按照需要逐行检索数据，而不是同时获取所有的数据并储存在客户端，可以用 mysql_use_result 代替 mysql_store_result：

```
#include <mysql/mysql.h>
MYSQL_RES *mysql_use_result(MYSQL *result);
```

这个函数也是得到一个连接对象并返回一个结果集合指针，或者在出错时返回 NULL。与 mysql_store_result 一样，这将返回一个结果集合对象。不过关键的是在返回时它实际上并没有将任何检索到的数据返回到结果集合中，而仅仅是将结果集合初始化来接收数据。为了检索数据，必须和反复调用 mysql_fetch_row，直到检索完所有的数据。

在使用 mysql_use_result 时，将不能使用函数 mysql_num_row、mysql_data_seek、mysql_row_seek、mysql_rows_tell。实际上这也不是非常严格：mysql_num_rows 可以被调用，但是在 mysql_fetch_result 检索完之前不能返回可用行的数量。这样它也就没什么太大用处了。但是使用 mysql_use_result 时，降低了网络通信流量负载，显著减少了客户的过量信息存储。对于较大的数据集合，用 mysql_use_result 逐行检索是较优越的一种方法。

例 9-6 用 mysql_use_result 检索数据。

```
1    /*ex5.c*/
2    #include <stdlib.h>
3    #include <stdio.h>
4    #include   <mysql/mysql.h>
5    MYSQL my_connection;
6    MYSQL_RES *res_ptr;
7    MYSQL_ROW sqlrow;
8    int main()
9    {
10       int res;
11       mysql_init(&my_connection);
12       if (mysql_real_connect(&my_connection, "localhost", "test","test", "testdb", 0, NULL, 0))
13        {
14              printf("Connection success\n");
15              res = mysql_query(&my_connection, "SELECT childno, fname,   age FROM
```

```
                                children
16                          WHERE age > 5");
17                 if (res)
18                         {
19                     printf("SELECT error: %s\n", mysql_error(&my_connection));
20                             return -2;
21                         }
22                         else
23                         {
24                                 res_ptr = mysql_use_result(&my_connection);
25                                 if (res_ptr)
26                                 {
27                                    printf("Retrieved %lu rows\n",(unsigned
                                           long)mysql_num_rows(res_ptr));
28                                    while ((sqlrow = mysql_fetch_row(res_ptr)))
29                                     printf("Fetched data...\n");
30                                    if (mysql_errno(&my_connection))
31                                     {
32                                            fprintf(stderr, "Retrive error:
                                                      s\n",mysql_error(&my_connection));
33                                            return -3;
34                                     }
35                         }
36                      mysql_free_result(res_ptr);
37                 }
38                 mysql_close(&my_connection);
39                         printf("Connection closed.\n");
40     }
41             else
42             {
43            fprintf(stderr, "Connection failed\n");
44            if (mysql_errno(&my_connection))
45                     {
46                 fprintf(stderr, "Connection error %d: %s\n",
47                 mysql_errno(&my_connection),mysql_error(&my_connection));
48                         return -1;
49               }
50            }
51      return 0;
52     }
```

说明：程序 6 同程序 5 基本相同，唯一的区别是用 mysql_use_result 函数代替了 mysql_store_result 函数(第 24 行)。从程序执行结果也可以看出 2 者之间的差别：

```
$ ./ex5
Connection success
Retrieved 0 rows
Fetched data...
Fetched data...
Fetched data...
Connection closed.
```

在得到结果以后并不能立即发现检索到的行数量。而且前面提到的检错技巧——mysql_errno(&my_connection)在错误不出现时将为零——更加易于应用。如果通过mysql_store_result编写代码，但是考虑到有可能需要回头转而使用mysql_use_result，可以在开始编码的时候记住这一点，然后通过检验所有函数返回的结果来保守地编码，这样会使此转换容易得多。

9.4.3.3 处理返回的数据

除非能够做进一步的工作，否则仅是检索数据是没有什么用处的。返回的数据有2种类型：

(1) 从数据库中检索到的实际信息。

(2) 关于数据的数据，所谓的“元数据”。

再确定列名称以及关于数据的其他信息，首先看一下怎样恢复数据并将数据打印。

可以使用mysql_field_cont函数，它得到一个连接对象并返回在结果集合中字段的数目：

```
#include <mysql/mysql.h>
unsigned int mysql_field_count(MYSQL *result);
```

这个函数也可以用于更普通的处理进程中，例如判断为何mysql_store_result调用失败。如果mysql_store_result返回NULL，但是mysq1_fleld_count返回一个大于0的数，可以知道结果集合中应该有一些列，但是在检索它们的时候出现了错误。另一方面，如果mysql_field_count返回0，则是没有检索到列，这就是为什么试图存储数据失败的原因。

这更多地适用于预先不知道SQL语句或者希望编写一个完全通用的查询进程模块的情况。

如果仅仅希望在非格式化的文本格式中得到结果信息，那么现在已经足以将数据直接打印出来，使用的是从mysql_fetch_row中返回的MYSQL_ROW结构体。可以添加一个非常简单的函数到例9-6程序中，来打印数据。

例9-7 打印查询的数据。

```
1    /*ex6.c*/
2    #include <stdlib.h>
3    #include <stdio.h>
4    #include   <mysql/mysql.h>
5    MYSQL my_connection;
```

```
6       MYSQL_RES *res_ptr;
7       MYSQL_ROW sqlrow;
8       void display_row(MYSQL *ptr);
9       int main()
10      {
11          int res;
12          mysql_init(&my_connection);
13          if (mysql_real_connect(&my_connection, "localhost", "test","test", "testdb", 0, NULL, 0))
14          {
15            printf("Connection success\n");
16            res = mysql_query(&my_connection, "SELECT childno, fname,    age FROM children
                  WHERE
17                                                          age > 5");
18              if (res)
19                    {
20                  printf("SELECT error: %s\n", mysql_error(&my_connection));
21                        return -2;
22              }
23              else
24              {
25                    res_ptr = mysql_use_result(&my_connection);
26                    if (res_ptr)
27                    {
28                       printf("Retrieved %lu rows\n",(unsigned long)mysql_num_rows(res_ptr));
29                       while ((sqlrow = mysql_fetch_row(res_ptr)))
30                       {
31                       printf("Fetched data...\n");
32                             display_row(&my_connection);
33                       }
34                        if (mysql_errno(&my_connection))
35                       {
36                           fprintf(stderr, "Retrive error: s\n",mysql_error(&my_connection));
37                           return -3;
38                         }
39                  }
40                  mysql_free_result(res_ptr);
41                 }
42                  mysql_close(&my_connection);
43                       printf("Connection closed.\n");
44           }
45          else
46          {
47            fprintf(stderr, "Connection failed\n");
```

```
48          if (mysql_errno(&my_connection))
49          {
50              fprintf(stderr, "Connection error %d: %s\n",
51              mysql_errno(&my_connection),mysql_error(&my_connection));
52              return -1;
53          }
54      }
55      return 0;
56  }
57
58  void display_row(MYSQL *ptr)
59  {
60      unsigned int field_count;
61      field_count=0;
62      while(field_count<mysql_field_count(ptr))
63      {
64          printf("%s ",sqlrow[field_count]);
65          field_count++;
66      }
67      printf("\n");
68  }
```

说明：程序 7 与程序 6 相比，添加了一个 display_row 函数，该函数的功能是显示查询出的数据(第 58~68 行)。在 main 函数的 while 循环中调用了 display_row 函数(第 32 行)。程序输出结果如下：

```
$ ./ex6
Connection success
Retrieved 0 rows
Fetched data...
1 Jenny 14
Fetched data...
2 John 10
Fetched data...
3 Jack 11
Connection closed.
```

尽管实现的功能很基础，不过程序毕竟已经能够执行了。我们没有考虑在结果中可能出现的 NULL 值。如果想要在表格中显示输出，例如在一个更加结构化的表单中得到数据以及关于数据的信息的话，那么该怎样做?

需要一次性获取一个字段的信息，然后把其输入到一个包含数据和源数据(关于返回数据的数据)的结构体中去。这需要通过 mysql_fetch_fleld 函数来实现：

```
#include <mysql/mysql.h>
MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result);
```

这个函数返回作为 MYSQL_FIELD 结构的一个结果集合的列的定义。重复调用这个函数在结果集合中检索所有关于列的信息。当没有剩下更多的字段时，mysql_fetch_field()返回 NULL。指向返回的字段结构体的指针可以用来访问储存在字段结构体中列的各种信息。字段内容由 mysql.h 所定义，如表 9-11 所示。

表 9-11　MYSQL_FIELD 结构体中的字段

字　段	说　明
char *name	列的名称，是一个字符串
char *table	列所在的表名称，这在选择使用多个表的时候更有用
char *def	如果调用 mysql_list_fields，则包含列的默认值
enum enum_field_types type	列的类型
unsigned int length	列的宽度，在定义表格的时候指定
unsigned int max_length	如果使用 mysql_store_result，则它包含找到的最长的实际列长度。如果使用 mysql_use_result，则不会对它进行设置
unsigned int flag	标识符。告知关于列的定义，而与实际找到的数据无关
unsigned int decimals	十进制数，只对数字字段有效

下面介绍一个非常有用的宏 IS_NUM，如果字段类型是数字形式的，则其返回值为真。这个宏如下：

```
if(IS_NUM(mysql_field_ptr->type)) printf("Numeric type field \n");
```

在更新程序以前，再介绍一个函数 mysql_field_seek：

```
#include <mysql/mysql.h>
MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result, MYSQL_FIELD_OFFSET offset);
```

此函数将字段光标设置到给定的偏移量。下一次调用 mysql_fetch_field 将检索与该偏移量关联的列的字段定义。若要定位于行的起始，则要传递一个值为 0 的 offset 值。

下面看一个例子。

例 9-8　结构化的数据库查询输出。

```
1    /*ex7.c*/
2    #include <stdlib.h>
3    #include <stdio.h>
4    #include   <mysql/mysql.h>
5    MYSQL my_connection;
6    MYSQL_RES *res_ptr;
```

```
7    MYSQL_ROW sqlrow;
8    void display_header();
9    void display_row(MYSQL *ptr);
10   int main()
11   {
12           int res;
13                   int first_row=1;
14           mysql_init(&my_connection);
15           if (mysql_real_connect(&my_connection, "localhost", "test","test", "testdb", 0, NULL, 0))
16           {
17                   printf("Connection success\n");
18                   res = mysql_query(&my_connection, "SELECT childno, fname,    age FROM children
19                                                WHERE age > 5");
20            if (res)
21                   {
22                   printf("SELECT error: %s\n", mysql_error(&my_connection));
23                           return -2;
24                   }
25                   else
26                   {
27                             res_ptr = mysql_use_result(&my_connection);
28                             if (res_ptr)
29                           {
30                                   display_header();
31                                   while ((sqlrow = mysql_fetch_row(res_ptr)))
32                                   {
33                                   if(first_row)
34                                           {
35                                                   display_header();
36                                                   first_row=0;
37                                           }
38                                           display_row(&my_connection);
39                                   }
40                                   if (mysql_errno(&my_connection))
41                                   {
42                                        fprintf(stderr, "Retrive error: s\n",mysql_error(&my_connection));
43                                        return -3;
44                                   }
45                      }
46                    mysql_free_result(res_ptr);
47                   }
48                   mysql_close(&my_connection);
49                           printf("Connection closed.\n");
```

```
50          }
51          else
52          {
53                  fprintf(stderr, "Connection failed\n");
54                  if (mysql_errno(&my_connection))
55                  {
56                                  fprintf(stderr, "Connection error %d: %s\n",
57                          mysql_errno(&my_connection),mysql_error(&my_connection));
58                                  return -1;
59                          }
60          }
61          return 0;
62  }
63
64  void display_header()
65  {
66          MYSQL_FIELD *field_ptr;
67          printf("Column details:\n");
68          while((field_ptr=mysql_fetch_field(res_ptr))!=NULL)
69          {
70                  printf("\t Name: %s\n",field_ptr->name);
71                  printf("\t Type: ");
72                  if(IS_NUM(field_ptr->type))
73                  {
74                          printf("Numeric field \n");
75                  }
76                  else
77                  {
78                          switch(field_ptr->type)
79                          {
80                                  case FIELD_TYPE_VAR_STRING:
81                                          printf("VARCHAR\n"); break;
82                                  case FIELD_TYPE_LONG:
83                                          printf("LONG\n");break;
84                                  default:
85                                          printf("Type is %d ,check in mysql_com.h\n",field_ptr->type);
86                          }
87                  }
88                  printf("\t Max width %d \n",field_ptr->length);
89                  if(field_ptr->flags & AUTO_INCREMENT_FLAG)
90                          printf("\t Auto increments \n");
91                  printf("\n");
92          }
```

```
93    }
94
95    void display_row(MYSQL *ptr)
96    {
97            unsigned int field_count;
98            field_count=0;
99            while(field_count<mysql_field_count(ptr))
100           {
101                   printf("%s ",sqlrow[field_count]);
102                   field_count++;
103           }
104           printf("\n");
105   }
```

说明：程序 8 在例 7 的基础上增加了一个 display_header 函数，用来输出相关的字段信息(第 64~93 行)，其中相关的类型定义在 mysql_com.h 文件中。

程序输出结果如下：

```
$ ./ex7
Connection success
Column details:
            Name: childno
            Type: Numeric field
            Max width 11
            Auto increments

            Name: fname
            Type: VARCHAR
            Max width 30

            Name: age
            Type: Numeric field
            Max width 11

Column details:
1 Jenny 14
2 John 10
3 Jack 11
Connection closed.
```

9.5 小　　结

在这一章中，介绍了 MySQL 的用法以及用 C 语言访问 MySQL 数据库的方法，读者应该掌握数据库的基本概念，MySQL 的安装和管理，MySQL 相关的实用命令以及基于 C 的 API 的编程方法。MySQL 提供的 API 函数很多，限于篇幅，不在这里一一介绍，读者可以在 MySQL 的官方网站 http://www.mysql.com 中找到相关的 API 介绍。

习　　题

一、填空题

1. 数据库语言一般可分为以下两种：一种是______，它具有语法简明、可独立使用等特点；另一种则嵌入到某种程序设计语言中．如 C、FORTRAN、PASCAL、COBOL 等，称为______。

2. 一个典型的关系型数据库通常由一个或多个被称作______的对象组成。

3. 在 SQL 命令中，______语句主要被用来对数据库进行查询并返回符合用户查询标准的结果数据；______语句被用来建立新的数据库表格；______语句向数据库表格中插入或添加新的数据行；______语句更新或修改满足规定条件的现有记录；______语句删除数据库表格中的行或记录；______语句删除某个表格以及该表格中的所有记录。

4. 对于密码为 1234 的用户 user1，为了用 mysql 开启选定的数据库 db1，我们需要输入______。

5. 在 MySQL 中，查看系统中当前存在的数据库命令是______，显示当前数据库中有哪些表的命令是______。

二、选择题

1. 所有的 SQL 语句在结尾处都要使用______符号。

   (A) ,　　(B) :　　(C) ;　　(D) .

2. 当向数据库表格中添加新记录时，在关键词 insert into 后面输入所要添加的______名称。

   (A) 数据库　　(B) 表格　　(C) 列　　(D) 值

3. ______是 MySQL 自带的主要管理实用程序。

   (A) mysqladmin　(B) mysqldump　(C) mysqlimport　(D) mysqlshow

4. 在用 C 语言访问 MySQL 数据库的程序中，______函数成功返回一个指向 MYSQL

结构的指针。

(A) mysql_init　(B) mysql_real_connect　(C) mysql_options　(D) mysql_error

5. 从 C 向一个 MySQL 数据库的连接包括______步。

(A) 1　(B) 2　(C) 3　(D) 4

三、上机题

1. 在 Linux 系统上安装 MySQL 数据库，并为根用户设置密码，密码自定义。

2. 在 MySQL 数据库中创建一个新数据库，并在数据库中创建一个数据库表 student：该表表示一个学校的学生登记表，表的内容包含 ID、学生姓名、年龄、入学日期、语文成绩、数学成绩、体育成绩、美术成绩。

3. 编写一个 C 程序，访问上题中新建的数据库，并向数据库中添加几条记录。

4. 编写一个 C 程序，访问上题中添加记录后的数据库，查询数据库中的内容，并用结构化方法输出查询内容。

CHAPTER 10

Linux 下的 GUI 编程

在这一章里，将学习如何编写能够运行在 Linux 图形化环境——X Window 系统(简称 X)里的程序。X Window 系统是在 Unix 类的操作系统中应用最为广泛的基于窗口的用户图形界面。它使用方便、界面直观、并且和具体的计算机硬件无关。同时它支持分布式的网络操作。所以基于 X Window 的应用程序一直在 Unix 类的操作系统中占有主导的地位。Linux 出现以后，X Window 系统也有了 Linux 系统上的实现，这就是 XFree86 系统。XFree86 比标准的 MIT 版的 X Window 支持更多的硬件。这样，它的应用就更加广泛。

本章将介绍以下内容：X 概念、Xlib 编程、GTK+/GNOME 编程以及 QT/KDE 编程。

10.1 概　　述

X Window 也称为 X 窗口系统，是由麻省理工学院(MIT)推出的窗口系统，简称 X。它旨在建立不依赖于特定硬件系统的显示窗口系统标准。X 窗口的最早商业版本是在 1986 年推出的 X10.4，该版本成为某些商业应用的基础。1987 年 9 月 MIT 发布了新版本 X11Rl，在 1988 年发布了 X11R2 版。版本 11 在它的速度、灵巧性和多屏幕的风格上都远远超过了 X10 版。它的出现标志着计算机工作站一个新时代的到来。X11 版本已经成为 Unix 平台的事实上的标准 GUI。现在，几乎所有工作站都采用了 X 窗口的标准，几乎所有工作站上的应用软件都采用了基于 X Window 的软件平台。目前 X 的版本到了 X11R6.4 版，它仍然是源代码发放的。同时，微机上的 X 系统也日益增多。例如，Linux 使用的 XFree86 就是基于 X11R6.4 的。

X 窗口系统主要由四个部分组成，下面将简要对它们进行介绍。

- X 服务器：与用户交互操作。
- X 协议：客户/服务器之间的通信。

- X 库：程序设计接口。
- X 客户：软件应用程序。

10.1.1　X 服务器

X 服务器，或者叫 X 显示服务器，是一个运行在软件用户计算机上的程序。它负责图形化显示硬件设备的控制工作，完成具体的输入和输出操作。 X 服务器响应来自 X 客户程序的请求，在屏幕上“画画”或者读取键盘或鼠标的输入。它负责传递输入数据以及向客户程序报告鼠标移动与按钮动作等事件。

10.1.2　X 协议

X 客户软件与 X 显示服务器之间的一切交互操作都必须通过交换消息才能进行。消息的类型和用途用法就构成了 X 协议。X 窗口系统特别有用的一个功能是：X 协议不仅能够穿越网络，就是对运行在同一台机器上的客户和服务器之间也同样适用。这就意味着即使用户手里只有一台功能较低的个人电脑或一台 X 终端(这是一种专为运行 X 服务器而设计和使用的机器)，也可以在更强大的联网计算机上运行各种 X 客户程序，而交互式操作和输出显示部分都是在自己的本地机器上进行的。

10.1.3　Xlib 库

只有那些实际编写 X 服务器的人才会真正对 X 协议感兴趣。大多数 X 软件都要使用一个 C 语言函数库作为程序设计的接口。这就是 Xlib 库，它为 X 协议里的信息交换提供了一个 API(应用程序设计接口)。Xlib 本身并没有增加太多的东西——它只能在屏幕上画线条和对鼠标动作做出响应。如果你需要菜单、按钮、卷屏条以及所有其他的东西，就必须自己编写它们。

从另一个方面看，Xlib 也没有强调任何特殊的 GUI 风格。它的作用只是一个中介，用户可以通过它创建出自己想要的风格。

10.1.4　X 客户

X 客户就是以后实际接触的软件程序，它们需要运行在某个计算机上，但是能够使用其他计算机上的显示和输入资源。它们通过向负责管理自己的 X 服务器提出对那些资源的访问请求而做到这一点。服务器一般都能够同时对来自许多客户的请求进行处理。它必须对键盘和鼠标在客户之间的使用情况进行裁决。客户程序使用 X 协议消息与服务器进行通信，而这些消息是通过 Xlib 函数来收发的。

10.2 Xlib 编程

可以看到X系统用一个通信协议在客户应用程序和X显示服务器之间划出了清晰的功能界线。基于 X 的应用软件是通过调用 X 的一系列 C 语言函数实现其各种功能的。这些函数称为 Xlib(X 库)，它提供了建立窗口、画图、处理用户操作事件等基本功能。Xlib 是一种底层库，用它来编与图形和交互界面程序虽然非常灵活，但比较复杂和繁琐，因此不准备在 Xlib 程序设计上多作停留，只是简略介绍一下它的基本编程方法。

下面从一个 X 的基本程序入手，简单介绍一下用 Xlib 编程的基本方法。该程序弹出一个窗口，并且在窗口的中央绘制“Hello world!”字符串。

典型的 X 应用程序在启动时必须对自己可能会用到的一切资源都进行初始化。它需要和 X 显示服务器建立起一个连接，选择使用的颜色和字体，然后在显示器上创建出一个窗口来。

客户程序连接和解除连接一个 X 服务器时要使用 XOpenDisplay 和 XCloseDisplay 函数。下面是它们的定义情况：

```
#include <X11/Xlib.h>
Display *XOpenDisplay(char *display_name);
void XCloseDisplay(Display *display);
```

display_name 参数指定的是我们打算连接的显示设备。如果它是 NULL，就使用环境变量 DISPLAY 的值。它的格式是“hostname:server[.display]”，一台主机可以有一个以上的 X 服务器，每个服务器又可以控制一个以上的显示设备。系统默认的显示设备通常就是“:0.0”，即本地计算机上第一个可用的服务器。如果想指定一个第二屏幕，比如桌面确实很大的时候，可以使用“:0.1”。

XOpenDisplay 返回的是一个 Display 结构，里面是刚才选择的 X 服务器的有关信息；如果没有 X 服务器可以被打开，就返回 NULL。只有在成功地从 XOepenDisplay 返回之后，客户程序才能开始使用 X 服务器。

当客户程序用完 X 服务器的时候，它必须以最初由 XOpenDisplay 调用返回的那个 Display 结构为参数调用 XCloseDisplay，这将清除该客户在显示设备上创建的一切窗口和其他资源。程序在退出前必须调用 XCloseDisplay，只有这样才能使排队中的错误得到报告。

接下来，可以根据 Display 获得屏幕的某些信息：

```
//获得默认的屏幕序号
int screen_num=DefaultScreen(display);
//获得屏幕的宽度和高度
int display_width=DisplayWidth(display, screen_num);
int display_height=DisplayHeight(display, screen_num);
```

获得了屏幕的相关信息后，下一步可以在屏幕上建立窗口，在 Xlib 中建立窗口的函数有如下 2 个：

```
Window XCreateSimpleWindows(
                    Display* display,               //display 参数
                    Window parent,                  //父窗口 ID，
                    int x,                          //x 坐标
                    int y,                          //y 坐标
                    unsigned int width,             //窗口宽度
                    unsigned int height,            //窗口高度
                    unsigned int border_width,      //边界宽度
                    unsigned long border,           //边界颜色
                    unsigned long background);      //背景颜色
Window XCreateWindow(
                    Display* display,               //display 参数
                    Window parent,                  //父窗口 ID，
                    int x,                          //x 坐标
                    int y,                          //y 坐标
                    unsigned int width,             //窗口宽度
                    unsigned int height,            //窗口高度
                    unsigned int border_width,      //边界宽度
                    unsigned long border,           //边界颜色
                    int         depth,              //颜色深度
                    unsigned int class,             //输入输出类型
                    Visual*    visual,              //Visual
                    Unsigned long valuemask,        //掩码
                    XSetWindowAttributes* attributes);  //窗口属性
```

这 2 个函数的参数比较多，每个参数的意义已经在参数后面进行了详细说明。这里使用前者建立窗口：

这里使用第 1 个函数来建立窗口：

```
win=XCreateSimpleWindows(display,               //display 参数
            RootWidow(display, screen_num) ,    //父窗口
            0, 0, width, height,                //位置和大小
            border_width,                       //边界宽度
            BlackPixel(display, screen_num),    //边界色
            WhitePixel(display, screen_num));   //背景色
```

其中，所建立的窗口的父窗口是根窗口，它可以用如下宏定义来获得：

```
RootWindow(display, screen_num)
```

接下来需要选择窗口感兴趣的事件类型：

```
XSelectInput(display,
             win,                        //窗口 ID
             ExposureMask                //曝光事件
             | KeyPressMask              //按键事件
             | ButtonPressMask           //鼠标按键事件
             | StructureNotifyMask);     //窗口改变事件
```

XSelectInput 可以为用户的 client 选取任何窗口上的 Event，只要该窗口有所需要的 Event 发生，X Server 就会送 Event 给 client。Xlib 采用了图元(GC,Graphics Context)作为写字符和画点线等基本元素的概念，在写字符时还应当指定字体，不过这里使用默认字体，因此不用建立字体结构。

```
//建立 GC
GC gc=XCreateGC(display,                 //Display
                win,                     //窗口 ID
                valuemask,               //GC 的掩码
                &values);                //获得 GCvalue 的值
```

在窗口建立后，它还只是存在内存中的一个数据结构，要显示窗口，需要把它映射到屏幕上：

```
XMapWindow(display, win);
```

至此，窗口和必要的 GC 已经建立，但是在这里不必要在窗口上画字符串，因为窗口会接收曝光(Expose)事件，在接收到该事件后再进行操作。下面进入事件循环，来顺序处理队列中的 X 事件。循环的基本规则是：

```
//进入事件循环
while(1) {
        XEvent    report;
        //取得队列中的事件
        XNextEvent(display, &report);
        switch (report.type) {
            case  Expose: …
            case  ConfigureNotiry: …
            case  ButtonPress: …
            case  KeyPress: …
            default     : …
        }
}
```

事件循环本身开始于 while 语句。驱动这个事件循环的函数调用是 XNextEvent。此函数的声明如下：

```
#include <X11/Xlib.h>
XNextEvetn(Display *display, XEvent *event_return);
```

其中，event_return 参数用于返回已经接收到的事件信息。

如果没有要处理的事件，XNextEvent 强制把一些缓冲的服务器请求写到服务器上。执行会在函数内部挂起，直到感兴趣的事件(这些是未屏蔽的事件)到达才会解挂。一旦接收到了感兴趣的事件，就会把事件信息复制到 event_return 参数指定的区域，而该函数将返回到调用程序。

XEvent 数据类型的定义是事件结构的一个大联合(union)。下面给出了 XEvent 定义的一个子集：

```
typedef   union _XEvetn{
  int                    type;        //事件类型
  XAnyEvent              xany;        //公共事件成员
  XKeyEvent              xkey;        //按键事件
  XButtonEvent           xbutton;     //鼠标按键事件
  XMotionEvent   xmotion;    //鼠标移动事件
  XExposeEvent   xexpose;    //窗口曝光事件
  XMappingEvent  xmapping;  //按键/按钮映射改变事件
  …
} XEvent;
```

XEvent 类型定义是它内部很多成员类型的联合。其中最基本的成员是 type，它指定了被描述事件的类型。

在屏幕上画字符串的函数如下：

```
XDrawString(
        Display* display,           // Display
        Drawable a,                      //  可画对象，如 Window
        GC gc,                      //  图元
        int x,                      // x
        int y,                      // y
        _Xconst char *string,       //  字符串指针
        int     length);            //  所画的长度，不一定是字符串长度
```

下面给出完整的程序。

例 10-1　使用 XLib 编程

```
1       /*ex1.c*/
2       #include <X11/Xlib.h>
3       #include <X11/Xutil.h>
4       #include <X11/Xos.h>
5       #include <X11/Xatom.h>
```

```
6       #include <stdio.h>
7
8       int main(int argc, char *argv[])
9       {
10      static char *string="Hello world!";          //要显示的字符串
11      Display *display;
12      int     screen_num;
13      Window    win;                               //窗口 ID
14      unsigned int width, height;                  //窗口尺寸
15      unsigned int border_width=4;                      //边界空白
16      unsigned int display_width, display_height;  //屏幕尺寸
17      int count;
18      XEvent     report;
19      GC    gc;
20      unsigned long valuemask=0;
21      XGCValues        values;
22      char *display_name=NULL;
23      //和 X 服务器连接
24      if((display=XOpenDisplay(display_name))==NULL)
25      {
26              printf("Cannot connect to X server %s\n",XDisplayName(display_name));
27              return -1;
28      }
29      //获得缺省的 screen_num
30      screen_num=DefaultScreen(display);
31      //获得屏幕的宽度和高度
32      display_width=DisplayWidth(display, screen_num);
33      display_height=DisplayHeight(display, screen_num);
34      //指定所建立窗口的宽度和高度
35      width=display_width/3;
36      height=display_height/4;
37      //建立窗口
38      win=XCreateSimpleWindow(display,
39                      RootWindow(display, screen_num),            //父窗口
40                      0,0,width, height,                        //位置和大小
41                      border_width,                             //边界宽度
42                      BlackPixel(display, screen_num),          //前景色
43                      WhitePixel(display, screen_num));         //背景色
44      //选择窗口感兴趣的事件掩码
45      XSelectInput(display, win, ExposureMask | KeyPressMask |
46                          ButtonPressMask | StructureNotifyMask);
47      //建立 GC
48      gc=XCreateGC(display, win, valuemask, &values);
```

```
49      //显示窗口
50       XMapWindow(display, win);
51      //进入事件循环
52       while(1)
53       {
54              //取得队列中的事件
55              XNextEvent(display, &report);
56              switch(report.type){
57              //曝光事件，窗口应重绘
58                      case Expose:
59                              //取得最后一个曝光事件
60                              if(report.xexpose.count!=0) break;
61                               //写字符串
62                              XDrawString(display, win, gc, width/2, height/2, string, strlen(string));
63                              break;
64                      //窗口尺寸改变，重新取得窗口的宽度和高度
65                      case ConfigureNotify:
66                              width=report.xconfigure.width;
67                              height=report.xconfigure.height;
68                              break;
69                      //鼠标点击或有按键，释放资源则退出
70                      case ButtonPress:
71                      case KeyPress:
72                              XFreeGC(display, gc);
73                              XCloseDisplay(display);
74                              return 1;
75                      default:
76                              break;
77              }
78          }
79      }
```

在编译 Xlib 程序时，要用到 X11 的库文件，因此编译命令如下：

```
$ gcc -o ex1 ex1.c -lX11
```

程序中各语句的功能已经在注释中说明，此处不再赘述。

下面是程序在 Ubuntu 下的运行结果，如图 10-1 所示。

最后总结一下 Xlib 编程的基本步骤：

(1) 打开 Display。

(2) 获取屏幕等基本信息。

(3) 建立窗口。

(4) 选择窗口等接收事件类型。

(5) 建立图元。

(6) 显示窗口。

(7) 进入事件循环。



图 10-1　程序 1 的运行结果

10.3　GTK+/GNOME 编程

从上一节可以看出，Xlib 的底层接口就像是 Microsft Windows SDK 开发工具包一样，它能对付相当复杂的程序，但不怎么出活。因此如果想又快又简单地编写出程序来，它并不是最佳的工具。

编写过这类 X1ib 程序的程序员肯定都希望找到一个更好的办法。而好办法确实有。常用的操作界面元素如按钮、滚动条和菜单等早就被实现过很多次了。X 窗口系统里的这类元素也叫做构件，把它们收集在一起就形成了人们所说的 X 工具包。其中，知名的 X 工具包包括以下几个：

(1) Xt。Xt 是在 X 的上面编写的一个免费函数库，它给 Xlib 库增加了一些功能，是一个能够简化应用程序设计的跳板。

(2) OpenLook。OpenLook 是 Sun 公司产品的一个免费的工具包，它强调了一种另类的观感。它是在一个名为 Xview 的函数库上面建立起来的，这个库与 Xt 很相似。

(3) Motif。Motif 是 OSF 组织的一个标准，设计目的是为 UNIX 桌面提供统一的观感。Motif 分为两个主要部分：一组用来定义 Xt 函数中使用的各种常数的头文件和一个用来简化对话框和菜单等元素的创建工作的易于使用的函数库。Motif 还定义了一种程序设计风格，不管程序员是否使用 Motif 工具包，都可以参照它来设计自己的程序。

(4) Qt。Qt 是一个由 Trolltech 公司出品的函数库，它构成了 KDE 桌面环境的基础，在大多数 Linux 发行版本里都能找到它。Qt 编程依赖于大量的 C++类集，通常其中每个类都

有一大批成员函数来处理类对象。

(5) GTK+。GTK+就是 GIMP 工具包，它是 GNOME 系统的基石。下面将详细介绍如何对这个高级环境进行程序设计。

10.3.1 GTK+/GNOME 简介

GNU 网络对象模型环境(GNU Network Object Model Environment)，即所谓的 GNOME，提供了功能强大而且易于使用的桌面环境，主要包括面板、桌面和一套用于组织程序接口的 GUI 工具。 它的目的并不仅仅是提供一个一致的界面，而且还提供了一个灵活的开发平台，用于开发具有强大功能的应用程序。

GNOME 在 GNU 公共许可(GNU Public License)下是完全免费的，没有任何限制。用户可以从 GNOME 的 web 站点 www.gnome.org 上直接获得其源代码。

GNOME 桌面的核心部件包括一个启动程序和桌面功能的面板。桌面上的其他部件由 GNOME 的兼容应用软件提供，例如文件管理器、web 浏览器和窗口管理器等。Ubuntu 的默认桌面环境就是 GNOME。GNOME 提供了 GNOME GUI 工具库，开发人员可以利用它来创建 GNOME 应用程序。使用符合 GNOME 标准的按钮、菜单和窗口的程序可以被称作是 GNOME 兼容的。

GTK+是 GNOME 应用软件使用的构件集，它的外观和感觉最初是来源于 Motif。构件集是一套可以用于桌面环境的 GUI 对象，按钮、窗口和工具栏等都是构件的实例。构件集设计用来支持功能性和灵活性。例如，按钮可以具有标签、图像或者是两者的任意组合。对象可以在运行时动态查询和修改。GTK+还包括一个主体引擎，用户可以使用这些构件来改变应用程序的外观。同时 GTK+构件保持了小巧和高效的特点。

GTK+构件集在库通用公共许可(LGPL, Library General Public License)下是完全免费的。LGPL 允许开发人员使用该构件集，并且像使用其他免费软件一样拥有其所有权。该构件集还有一个特点，就是它支持许多编程语言，包括 C、C++、Perl、Python 等。同时，它完全支持国际化，允许基于 GTK+的应用程序用于其他的字符集，例如亚洲语言。

在 GNOME 上运行的程序本质上都是包含 GNOME 和 GTK+函数的 C 程序。GNOME 和 GTK+函数为上述程序处理 GNOME 桌面操作。当使用 GNOME 编程时，会用到大量的函数和结构，这些函数和结构包含在许多库里，组成了 GNOME 应用软件的不同部分。要获取这些库中包含的有关函数、定义和结构的详细信息，建议使用 GNOME 开发者 web 站点 http://developer.gnome.org 上丰富的文档资源。这些文档包括详细的指南、手册和参考书，其中还有 GNOME、GTK 和 GOK API 的参考大全。

本书只是提供了关于这些库以及如何利用它们来编制 GNOME 程序的一个概要的纵览。在 GNOME 中编程仅仅需要一些基本函数就可以编制简单的用户界面。可以将 GTK+函数看作是低级的操作，而将 GNOME 函数看作是易于使用的高级操作。GNOME 函数经常合并几个 GTK+函数，使 GUI 任务相对易于编程。GNOME 程序本质上是含有 GTK+ 函

数和 GNOME 函数的 C 程序。因为几个基本的 GNOME 操作是由 CTK+函数来操纵的，本书首先讨论基本的 GTK+编程，然后讨论 GNOME 编程。

GNOME 库提供了 GNOME 应用软件中使用的最高级函数。在它们的下面是 GTK+库。GTK+是为 GNU 图像处理程序(GIMP，GNU Image Manipulation Program)开发的工具箱。GTK+由 GIMP 工具箱 GTK(GIMP Toolkit)和 GIMP 绘图工具箱 GDK(GIMP Drawing Kit)库组成。

GTK 包含管理构件和用户界面任务的函数以及结构。这些函数和结构可以在任何 GNOME 程序中直接访问。实际上，GNOME 应用软件就是一个具有 GNOME 库函数的 GTK 程序。GTK 是高层的库函数，它基本上不使用 Xlib 库函数，而是使用更底层的函数库 GDK 和界面无关的函数库 Glib。

GDK 是底层的图形函数库，它包含 GTK 所使用的基本图形操作函数，比如基本图元、颜色、事件处理、图像和位图、窗口、拖放函数等。

Glib 是一些与界面无关的函数构成的基本库，它定义了诸如基本类型、内存操作、定时器、字符串操作、输入输出操作、线程等系列的函数以及一些宏定义等。

以上库函数的结构如图 10-2 所示。

GNOME Application
GTK
GDK/Glib
Xlib

图 10-2　GNOME 程序库函数结构

10.3.1.1　GTK+

GTK+包含大量函数，用于各种不同类型的构件集，例如菜单、按钮和窗口等。它也支持将相应的 GUI 事件(如点击鼠标)绑定到对象(如按钮)。浏览 web 站点 www.gtk.org 和 developer.gnome.org 可以查阅 GTK API 的在线文档。这些文档包括一个所有 GTK 函数的列表以及关于 GTK 编程的详细指南。强烈建议利用好这些文档——由于篇幅所限，本书只能对此进行简单介绍，列出一部分常用的 GTK 函数。同时请参阅 GTK 的头文件，那里面有关于不同函数和结构的详细声明，包括它们的参数和返回值。

10.3.1.2　GNOME 库

GNOME 应用软件具有同样的 GUI 界面，同样的外观和感觉，而正是 GNOME 库使上述特点成为可能。尽管 GNOME 应用软件是使用 GNOME 库函数的 GTK 程序，GNOME 库还提供了一些高级构件和一些简单操作，而这些并没有包含在 GTK+库中。表 10-1 列出了 GNOME 库。

表 10-1　GNOME 库

库	说　明
libaudiofile	读取多种音频文件格式，例如：AIFF、AIFC、WAV
libgdk_imlib	加载多种不同文件格式的函数，例如：JPEG、GIF、TIFF、PNG、BMP 等
libgtk	这是 GTL 工具箱库，GNOME 应用软件的所有 GUI 元素都是用 libgtk 写的，这些元素包括按钮、菜单、滚动条等
libgnome	包含 GNOME 桌面环境的实用程序，例如配置、帮助、管理会话的程序。本库独立于任何 GUI 工具箱
libgnomeui	包含 GTK+构件集的工具箱扩展，用于创建对话框和消息框、菜单栏、工具栏、状态栏等
liggnorba	使用 ORBit(GNOME 实现的 CORBA)的一个库
libzvt	包含终端构件的一个库
libart_lgpl	包含用于 GnomeCanvas 的图形函数

libgnome 和 libgnomeui 是任何 GNOME 应用软件所需要的两个主要库。

libgnome 是一个函数集，设计用来独立于任何特定的 GUI 工具箱。这些函数可以用于任何种类的函数，不论该函数是一个命令行界面或者是没有界面。这些函数是独立于任何特定的 GUI 工具箱的。

libgnomeui 库包含提供 GUI 界面的函数。这些函数被绑定到特定的 GUI 工具箱，例如 GTK。编程人员可以用它轻松地创建对话框和消息框以及菜单栏、工具栏和状态栏等。这个库提供了大量的图标，编程人员可以将其用在对话框、菜单入口和按钮中。因为所有 GNOME 应用软件都使用 libgnomeui 来创建这些通用的 GUI 元素，所以就保证了视觉上的一致性。

10.3.2　GTK+编程

任何 GTK 程序都需要几个基本的函数和组成部分。首先需要至少包含头文件 gtk.h。根据所使用的构件和函数，可能会需要其他的 GTK 头文件。然后必须为想要使用的构件定义指针。随后需要利用 gtk_initt 函数初始化 GTK 库。做完这些以后，就可以利用 GTK 函数定义构件，并将它们的地址赋给开始定义的指针。然后，可以利用 GTK 函数为构件指定动作和属性，如显示它们。例如，一个关闭方框(Close box)的事件 delete_event 绑定到窗口和函数 gtk_main_quit。因此，如果一个用户点击了窗口中的关闭方框(Close box)，程序结束。最后，利用 gtk_main 函数运行构件。

例 10-2　下面的程序定义了一个简单的 GTK+程序，显示了一个简单窗口。

```
1     /*ex2.c*/
2     #include <gtk/gtk.h>
```

```
3       #include <stdio.h>
4
5       int main(int argc, char *argv[])
6       {
7               GtkWidget *window1;
8               gtk_init(&argc, &argv);
9               window1=gtk_window_new(GTK_WINDOW_TOPLEVEL);
10              gtk_signal_connect(GTK_OBJECT(window1),"delete_event",
                                    GTK_SIGNAL_FUNC(gtk_main_quit), NULL);
11              gtk_widget_show(window1);
12              gtk_main();
13              return 0;
14      }
```

说明：在上面的例子中，头文件 gtk.h 包含 GTK 变量、宏和函数的定义(第 2 行)。window1 是结构 GtkWidget 定义的指针(第 7 行)。实际指向的结构将稍后由函数确定，该函数用来创建一个给定的结构。gtk_init 函数生成初始化设置，例如默认的视觉和颜色图，并初始化 GTK 库，检查 GTK 参数(第 8 行)。函数 gtk_window_new 创建一个新的窗口结构并返回它的地址，该地址随后被赋给窗口指针(第 9 行)。现在窗口指向 GTK 窗口结构。参数 GTK_WINDOW_TOP_LEVEL 将窗口控制权交给窗口管理器，利用窗口管理器的默认设置来显示窗口。gtk_signal_connect 函数的作用在后面详细说明(第 10 行)。然后函数 gtk_widget_show 显示该窗口——注意窗口指针是用来作为该函数的参数的(第 11 行)。最后，函数 gtk_main 开始交互过程，等待事件的出现，例如按钮选择或者是鼠标单击。

利用 gcc 编译器和 GTK+库来编译 GTK+程序。要在命令行中指定 GTK+库，可以使用 gtk-config 脚本命令。这个命令确定了编译 GTK+程序所需的编译器选项。示例如下：

```
`gtk-config --cflags --libs`
```

gtk-config 是一个需要在命令行中运行的程序。要运行它，将该命令和它的参数都加上反引号。反引号是一个 shell 操作符，它用来在命令行中运行引起来的命令，并将返回值放在该命令行的同样位置。可以将这个操作理解为一个宏功能，用返回值取代运行的命令。在上述示例中，gtk-config 命令及其参数 cflags 和 libs 将把所需要的编译器 GTK 标志(flags)和库放在 gcc 命令的命令行中。然后 gcc 命令利用这些标志和库运行。上述程序的编译命令如下：

```
gcc -o ex2 ex2.c `gtk-config --cflags --libs`
```

下面是程序在 Ubuntu 下的运行结果，如图 10-3 所示。可以对这个窗口进行移动、尺寸调整和关闭等操作。



图 10-3　例 10-2 程序的运行结果

10.3.2.1　数据类型

GTK 定义有自己的基本数据类型集合，他们中的大多数都有直接对应的 C 语言标准数据类型。这使得不同计算机平台之间的代码移植工作更容易实现；并且在某些情况下——比如用 gpointer 替 void *的情况下，它可以改进程序的可读性，使之更容易理解。坚持使用这些新的数据类型可以保证我们的代码即使在其底层实现发生了变化的情况下仍然能够继续工作。GTK 中的常用数据类型集合如表 10-2 所示。

表 10-2　GTK 数据类型

GTK 数据类型	对应的 C 语言类型
gchar	char
gshort	short
glong	long
gint	int
gboolean	char
gpointer	void *

10.3.2.2　信号和事件

GNOME 程序的运作类似于其他的 GUI 程序——它是由事件驱动的。在事件驱动程序中，首先定义对象，用户可以对该对象进行操作。然后启动交互式函数，连续地检查特定事件，例如一次鼠标单击或者是菜单选择。如果检测到这样的一个事件，它就被传递给相应的函数进行处理。例如，如果用户单击一个 OK 按钮，该鼠标单击被检测到并被传递给一个函数，而该函数正是用来处理一次单击 OK 按钮的操作的。当函数结束后，它将控制交回到交互程序。

GTK 的操作更加复杂一些。当事件在某个构件上发生时，该构件将发出一个信号(signal)，然后该信号被用于运行一个函数，这个函数与信号和对象都是关联的。例如，当用户单击了一个Close 按钮，Close 按钮构件检测到鼠标单击事件并发出一个 clicked 信号(表示被单击了)。该信号被检测到后，运行与它关联的函数。

也可以将一个事件直接关联到一个函数，为此，必须将给定对象的信号与特定的函数相关联，与特定对象关联的函数一般称为“处理机”(handlers)或者“回调函数”(callbacks function)。一个信号发出后，其处理机或者回调函数就被调用。这个过程称为“发射”(emission)。注意，这里提到的信号与本书前面讲到的 Linux 系统中使用的信号没有任何共同之处。

要将一个特定的事件与函数关联，且该函数将根据给定的信号运行，可以利用函数 gtk_signal_connect 或者 gtk_signal_connect_object。检测到信号后，它的关联函数自动运行。gtk_signal_connect 函数用来调用要传递参数的函数，gtk_signal_connect 或 gtk_signal_connect_object 函数都可以用来调用不需要参数的函数。下面是 gtk_signal_connect 函数的语法：

```
#include <gtk/gtk.h>
gint gtk_signal_connect(GtkObject *object, gchar *name, GtkSignalFunc func, gpointer func_data );
```

其中，object 是定义的对象 GtkObject 指针，例如一个按钮；name 是信号名字指针，例如一个鼠标单击；func 是对象事件发生时要运行的函数；func_data 是传递给该函数的任何参数。

当检测到指定对象的一个信号时，该信号关联的回调函数被调用并被执行，语法如下所示：

```
void callback_func(GtkWidget *widget, gpointer callback_data);
```

因此要将一次单击按钮的事件与 hello 函数关联，需要利用下面的 gtk_signal_connect 语句：

```
gint gtk_signal_connect(GTK_OBJECT(mybutton),"clicked",GTK_SIGNAL_FUNC(hello), NULL);
```

在上述语句中，对象是 mybutton，clicked 是单击信号，而 hello 是函数，当检测到信号时该函数运行。GTK_OBJECT 和 GTK_SIGNAL_FUNC 是进行类型检查的宏命令，它保证对象以适当的类型传递。

某些对象具有与其关联的信号。例如，按钮对象可以与 clicked 信号或者 enter 信号关联。当用户按下然后释放鼠标键时发出 clicked 信号，而当用户移动鼠标指针到按钮对象上面时发出 enter 信号。按钮信号如表 10-3 所示。

表 10-3　GTK 按钮信号

信　　号	说　　明
pressed	当鼠标指针放置在按钮上时，按下鼠标键
released	当鼠标指针放置在按钮上时，释放鼠标键
clicked	当鼠标指针放置在按钮上时，按下并释放鼠标键
enter	鼠标指针移动到按钮上面
leave	鼠标指针离开按钮

也可以利用信号连接函数将事件直接与一个对象和函数关联起来，而不必使用信号。事件是由 X11 服务器传输的消息，用来指示如鼠标单击和菜单选择等发生的事情。在 gtk_signal_connect 函数中，可以使用事件名而不是信号名，在示例程序 2 中，使用的就是事件名"delete_event"。事件回调函数包含一个为事件增加的参数。这个参数的类型可以是 GdkEvent 或者是其他几个事件类型之一。

```
void callback_func(GtkWidget *widget, GdkEvent *event, gpointer callback_data);
```

例如，要将事件 button_press_event 与一个 OK 按钮关联。可以使用"button_press_event"作为信号名。下面的例子将按钮事件 button_press_event 与函数 button_press_callback 关联：

```
gint gtk_signal_connect(GTK_OBJECT(button),"
      button_press_event",GTK_SIGNAL_FUNC(button_press_callback), NULL);
```

信号连接回调函数——在本例中即 button_press_callback 函数，使用事件类型 GdkEventButton 作为它的事件参数。

```
static gint button_press_callback(GtkWidget *widget, GdkEventButton *event, gpointer callback_data);
```

表 10-4 所示列出了一些常见的 GTK 事件。

表 10-4　常见的 GTK 事件

事　件	GtkWidget 信号
GDK_DELETE	"delete_event"
GDK_DESTROY	"destroy_event"
GDK_EXPOSE	"expose_event"
GDK_MOTION_NOTIRY	"motion_notify_event"
GDK_BUTTON_PRESS	"button_press_event"
GDK_2BUTTON_PRESS	"button_press_event"
GDK_3BUTTON_PRESS	"button_press_event"
GDK_BUTTON_RELEASE	"button_release_event"
GDK_KEY_PRESS	"key_press_event"
GDK_KEY_RELEASE	"key_release_event"
GDK_ENTER_NOTIFY	"enter_notify_event"
GDK_LEAVE_NOTIFY	"leave_notify_event"

在例 10-2 的基础上看一个稍微复杂的例子。

例 10-3　该程序弹出一个窗口，并且在窗口中显示一个按钮。

```
1      /*ex3.c*/
2      #include <gtk/gtk.h>
```

```
3
4    void hello(GtkWidget *widget, gpointer data)
5    {
6            g_print("Hello World!\n");
7    }
8
9    gint delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
10   {
11           g_print("delete event occurred!\n");
12           return TRUE;
13   }
14
15   void destroy(GtkWidget *widget, gpointer data)
16   {
17           gtk_main_quit();
18   }
19   int main(int argc, char *argv[])
20   {
21           GtkWidget *window;
22           GtkWidget *button;
23
24           gtk_init(&argc, &argv);
25           window=gtk_window_new(GTK_WINDOW_TOPLEVEL)      ;
26           gtk_signal_connect(GTK_OBJECT(window),"delete_event",
                             GTK_SIGNAL_FUNC(delete_event), NULL);
27           gtk_signal_connect(GTK_OBJECT(window),"destroy", GTK_SIGNAL_FUNC(destroy),
                             NULL);
28
29           gtk_container_set_border_width(GTK_CONTAINER(window),10);
30           button=gtk_button_new_with_label("Hello World!");
31
32           gtk_signal_connect(GTK_OBJECT(button),"clicked", GTK_SIGNAL_FUNC(hello),
                             NULL);
33
34           gtk_signal_connect_object(GTK_OBJECT(button),"clicked",
                 GTK_SIGNAL_FUNC(gtk_widget_destroy),GTK_OBJECT(window));
35               gtk_container_add(GTK_CONTAINER(window), button);
36
37          gtk_widget_show(button);
38
39          gtk_widget_show(window);
40          gtk_main();
41   }
```

说明： 与例 11-2 类似，程序首先调用 gtk_init 函数进行初始化，然后调用 gtk_window_new 函数建立新窗口，并把 delete_event 事件与函数 delete_event 关联起来，把 destroy 事件和 destroy 函数关联起来(第 24~27 行)。当单击关闭窗口的按钮时，将调用 delete_event 函数，在 delete_event 函数中，打印输出"delete event occurred"信息，如果返回 FALSE，GTK 将发出"destroy"信号，如果返回 TRUE，则不让窗口关闭(第 9~13 行)。然后是设置窗口的边界宽度(第 29 行)，建立一个标签是"Hello World"的按钮(第 30 行)，当按钮被单击时，即接收到"clicked"信号，将调用 hello 函数(第 32 行)。在 hello 函数中，往终端输出"Hello World"信息(第 4~7 行)。同时按钮被单击时也将调用 gtk_widget_destroy 函数关闭窗口(第 34 行)。下面是把按钮加入顶级窗口中(第 35 行)，显示按钮(第 37 行)，显示顶级窗口(第 39 行)，进入事件循环(第 40 行)。

程序在 Ubuntu 下运行结果如图 10-4 所示。

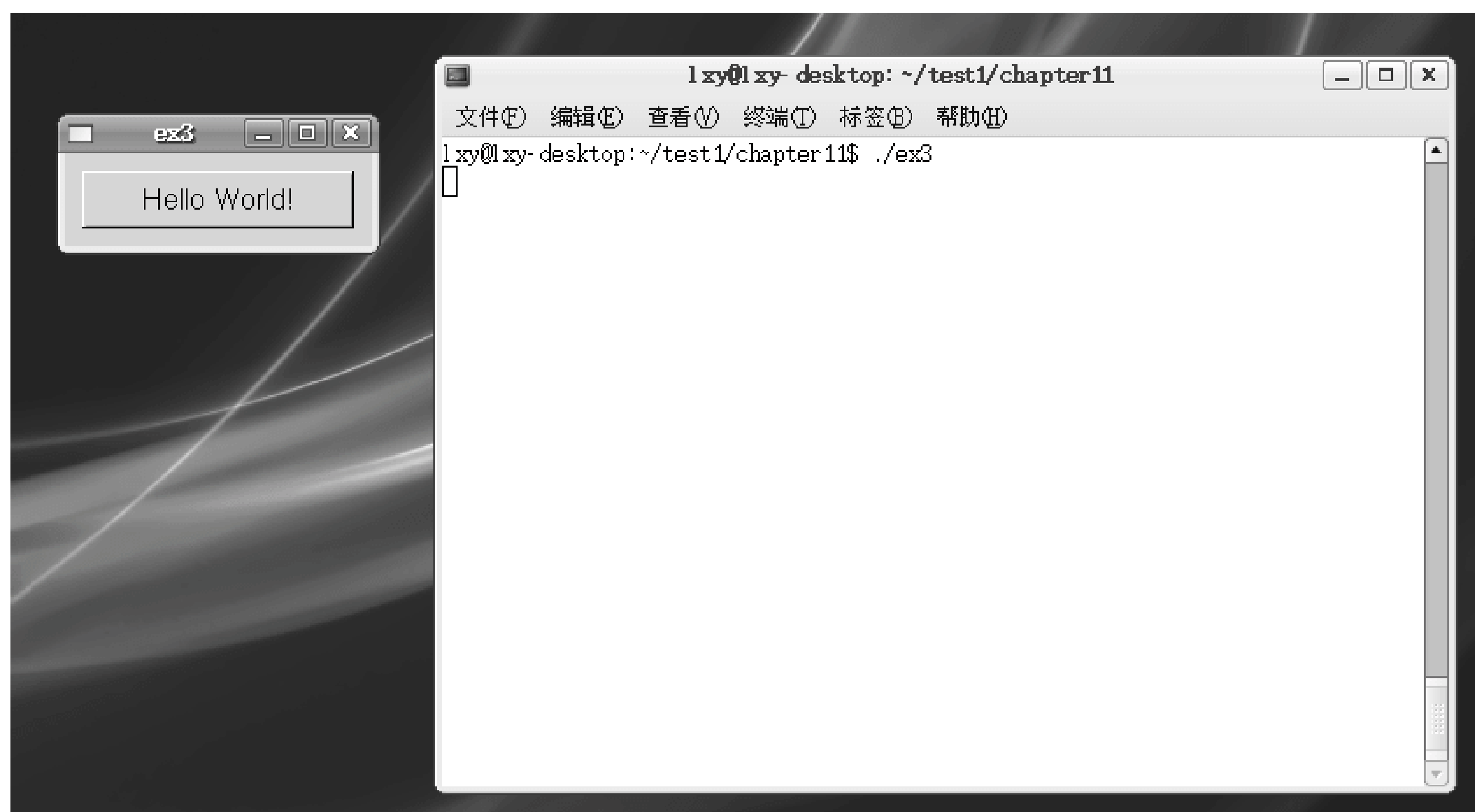


图 10-4　例 10-3 运行结果

当单击窗口的关闭按钮时，在终端窗口上输出"delete event occurred"信息并不退出，而当单击按钮时，将在终端窗口上显示"Hello World"后退出。

下面总结一下编写 GTK+程序的基本步骤：

(1) 初始化。

(2) 创建主窗口。

(3) 创建并加入子窗口。

(4) 设置构件回调函数。

(5) 显示窗口。

(6) 进入事件循环。

上面的例子是英文编程的基本例子，如果在程序中使用中文，还应对上面的例子作适当的修改。当然最基本的要求还是 Linux 系统必须有一个正确的设置。如果读者用的是 Ubuntu 系统，在/etc/gtk/目录下存在 gtkrc.zh_CN 文件，需要把它们改名为 gtkrc.zh_CN.utf-8。命令如下：

```
$sudo cp /etc/gtk/gtkrc.zh_CN /etc/gtk/gtkrc.zh_CN.utf-8
```

如果不存在，直接建立

```
$sudo gedit /etc/gtk/gtkrc.zh_CN.utf-8
```

文件内容为：

```
# $(gtkconfigdir)/gtkrc.zh_CN
#
# This file defines the fontsets for Chinese language (zh) using
# the simplified chinese standard GuoBiao as in mainland China (CN)
#
# 1999, Pablo Saratxaga < pablo@mandrakesoft.com >
#
style "gtk-default-zh-cn" {
fontset = "-adobe-helvetica-medium-r-normal--12-*-*-*-*-*-iso8859-1,\
-*-*-medium-r-normal--16-*-*-*-*-*-gb2312.1980-0,*-r-*"
} class "GtkWidget" style "gtk-default-zh-cn"
```

使程序支持中文非常简单，只需要在初始化 gtk 之前调用 gtk_set_local 设置函数即可：

```
gtk_set_locale();
 gtk_init(&argc, &argv);
…
```

将在下面的程序中看到使用中文的例子。

10.3.2.3 GTK+布局

在上一节小已经接触到了 GTK+的基本构件 button，其实 GTK+有几十个类似于按钮的基本构件，它们在建立起来以后便可以遵照某些布局规则组合成一个真正使用的程序。一般程序只使用水平和垂直方向的布局就可以了。

1. 布局基本示例

这里将从简单的例子入手，探讨 GTK 的布局。布局的目的是按程序员所希望的方式生成界面，并且当用户改变窗口大小时界面不会有大的变动。下面的例子是生成一个简单的提示用户输入的程序，它包含 GTK 的两个基本构件：标签(Label)和输入区(Entry)，两个构件水平排列。

例 10-4　GTK+程序布局示例。

```
1    /*ex4.c*/
2    #include <gtk/gtk.h>
3    #include <stdio.h>
4
```

```
5
6     gint delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
7     {
8         g_print("delete event occurred!\n");
9         return TRUE;
10    }
11
12    void destroy(GtkWidget *widget, gpointer data)
13    {
14        gtk_main_quit();
15    }
16    int main(int argc, char *argv[])
17    {
18        GtkWidget *window;
19        GtkWidget *hbox;
20        GtkWidget *label, *entry;
21
22        gtk_set_locale();
23        gtk_init(&argc, &argv);
24        window=gtk_window_new(GTK_WINDOW_TOPLEVEL)    ;
25        gtk_signal_connect(GTK_OBJECT(window),"destroy", GTK_SIGNAL_FUNC(destroy),
                      NULL);
26
27        gtk_container_set_border_width(GTK_CONTAINER(window),10);
28        hbox=gtk_hbox_new(FALSE,0);
29
30        gtk_container_add(GTK_CONTAINER(window), hbox);
31        gtk_widget_show(hbox);
32
33        label=gtk_label_new("请输入:");
34        gtk_box_pack_start(GTK_BOX(hbox), label, FALSE, FALSE,0);
35        gtk_widget_show(label);
36
37        entry=gtk_entry_new();
38        gtk_box_pack_start(GTK_BOX(hbox), entry, TRUE, TRUE, 0);
39        gtk_widget_show(entry);
40
41        gtk_widget_show_all(window);
42        gtk_main();
43        return 0;
44    }
```

说明：上例中使用了用于安排构件的 GTK+构件 hbox，它的作用仅用于负责子窗口的

布局，它是不可见的(第 28 行)。它的第一个参数用来控制它的子构件所占的空间是否具有相同的高度或宽度(这里是指宽度)。如果设置为 TRUE，两个子构件 label 和 entry 将具有相同的宽度。它的第二个参数是构件之间的间隔。hbox 建立后，加入到顶级窗口 window 中(第 30~31 行)，然后建立标签 label，并把它安排在 hbox 内(第 33~35 行)。其中，gtk_box_pack_start 函数用于安排子组件，它的使用形式是：

```
void gtk_box_pack_start(GtkBox *box, GtkWidget *child, gint expand, gint fill, gint padding);
```

其中，expand 如果为 TRUE，则组件填满 hbox 的所有空白，否则 hbox 缩到组件大小。fill 如果设置为 TRUE，则组件将填充组件之间的空间，否则 hbox 在构件周围产生空白区域。这只有在 exand 为 TRUE 时才会起作用。最后一个参数是构件旁边的空白。

对于输入条 entry，处理的方法相同(第 37~39 行)，只要把 entry 设置为可扩展的，并且填充 hbox 剩余的空间，在缩放窗口时，entry 始终水平地充满窗口。

程序在 Ubunt 下的运行结果如图 10-5 所示。

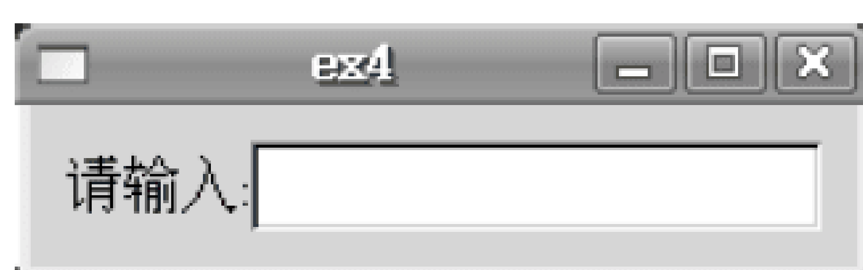


图 10-5　程序 4 的运行结果

2. 更高级的布局

下面介绍同时使用水平和垂直的布局构成更高级的用户界面。

例 10-5　界面总体布局是垂直方向的，它容纳了四个构件：三个水平容器和一个窗格构件，在水平构件中又容纳了其他构件。

```
1      /*ex5.c*/
2      #include <gtk/gtk.h>
3      #include <stdio.h>
4
5
6      void destroy(GtkWidget *widget, gpointer data)
7      {
8              gtk_main_quit();
9      }
10
11     void callback_ok(GtkWidget *widget, gpointer data)
12     {
13             printf("确定按钮被按下\n");
14             gtk_main_quit();
15     }
16
17     void callback_cancel(GtkWidget *widget, gpointer data)
```

```
18	{
19		printf("取消按钮被按下\n");
20		gtk_main_quit();
21	}
22	int main(int argc, char *argv[])
23	{
24		GtkWidget *window;
25		GtkWidget *vbox, *hbox1, *hbox2, *hbox3;
26		GtkWidget *label1, *label2;
27		GtkWidget *entry1, *entry2;
28		GtkWidget *sep;
29		GtkWidget *button_ok, *button_cancel;
30	
31		gtk_set_locale();
32		gtk_init(&argc, &argv);
33	
34		window=gtk_window_new(GTK_WINDOW_TOPLEVEL)	;
35		gtk_signal_connect(GTK_OBJECT(window),"destroy", GTK_SIGNAL_FUNC(destroy),
			NULL);
36	
37		gtk_container_set_border_width(GTK_CONTAINER(window),10);
38	
39		vbox=gtk_vbox_new(FALSE,10);
40		gtk_container_add(GTK_CONTAINER(window), vbox);
41		gtk_widget_show(vbox);
42	
43		hbox1=gtk_hbox_new(FALSE,5);
44		gtk_box_pack_start(GTK_BOX(vbox), hbox1, FALSE, FALSE, 0);
45		gtk_widget_show(hbox1);
46	
47		label1=gtk_label_new("姓名:");
48		gtk_box_pack_start(GTK_BOX(hbox1), label1, FALSE, FALSE,0);
49		gtk_widget_show(label1);
50	
51		entry1=gtk_entry_new();
52		gtk_box_pack_start(GTK_BOX(hbox1), entry1, TRUE, TRUE, 0);
53		gtk_widget_show(entry1);
54	
55		hbox2=gtk_hbox_new(FALSE,5);
56		gtk_box_pack_start(GTK_BOX(vbox), hbox2, FALSE, FALSE, 0);
57		gtk_widget_show(hbox2);
58	
59		label2=gtk_label_new("电话号码:");
```

```
60          gtk_box_pack_start(GTK_BOX(hbox2), label2, FALSE, FALSE,0);
61          gtk_widget_show(label2);
62
63          entry2=gtk_entry_new();
64          gtk_box_pack_start(GTK_BOX(hbox2), entry2, TRUE, TRUE, 0);
65          gtk_widget_show(entry2);
66
67          sep=gtk_hseparator_new();
68          gtk_box_pack_start(GTK_BOX(vbox), sep, FALSE, FALSE, 0);
69          gtk_widget_show(sep);
70
71          hbox3=gtk_hbox_new(FALSE,5);
72          gtk_box_pack_start(GTK_BOX(vbox), hbox3, FALSE, FALSE, 0);
73          gtk_widget_show(hbox3);
74
75          button_ok=gtk_button_new_with_label("确定");
76          gtk_box_pack_start(GTK_BOX(hbox3), button_ok, TRUE, FALSE,0);
77          gtk_signal_connect(GTK_OBJECT(button_ok), "clicked",
                            GTK_SIGNAL_FUNC(callback_ok), entry1);
78          gtk_widget_show(button_ok);
79
80          button_cancel=gtk_button_new_with_label("取消");
81          gtk_box_pack_start(GTK_BOX(hbox3), button_cancel, TRUE, FALSE,0);
82          gtk_signal_connect(GTK_OBJECT(button_cancel), "clicked",
                            GTK_SIGNAL_FUNC(callback_cancel), NULL);
83          gtk_widget_show(button_cancel);
84
85
86          gtk_widget_show_all(window);
87          gtk_main();
88          return 0;
89      }
```

说明：与例 10-4 不同的是，程序首先建立了一个垂直容器，用来垂直安排构件和水平的容器(第 39~41 行)，然后建立第 1 个水平容器，容纳一个标签和输入条(第 43~45 行)，建立第 1 个标签和输入条(第 47~53 行)，随后建立第 2 个水平容器，容纳一个标签和输入条(第 55~57 行)，接着建立第 2 个标签和输入条(第 59~65 行)。在建立第 3 个水平容器之前，加入分隔构件(第 67~69 行)，接着建立第 3 个水平容器，容纳 2 个按钮(第 71~73 行)。“确定”和“取消”按钮分别建立，并把单击它们的事件与回调函数关联(第 75~83 行)。

程序在 Ubuntu 桌面环境下执行结果如图 10-6 所示。



图 10-6　程序 5 的执行结果

当单击“确定”按钮时，程序在终端输出“确定按钮被按下”信息后退出，单击“取消”按钮时，则输出“取消按钮被按下”信息后退出。

3. 总结

下面总结一下通用的创建和使用构件的过程：

(1) 使用 gt_*_new 来创建构件。

(2) 把构件和信号关联起来。

(3) 把构件排列在容器之中，一般使用 gtk_container_add 和 gtk_box_pack_start 两种方法。

(4) 使用 gtk_widget_show 显示构件。

在上面的例子中已经接触到了 GTK+的几个基本构件：按钮(Button)、标签(Label)、输入条(Entry)和分隔构件(Separator)。GTK+有几十个类似的基本构件，限于篇幅，在此不再一一介绍，读者可以去 GTK+的网站上查阅相关的资料和它们的使用方法。

10.3.3　使用 GTK+编写 GNOME 程序

GNOME 程序构筑在 GTK+程序之上，它提供了 GNOME 函数，可以使你更轻松地创建与 GNOME 桌面风格一致的 GNOME 界面。要创建一个简单的 GNOME 程序，首先为 GNOME 构件定义 GTK 对象，然后利用 GNOME 函数来初始化程序和定义用户的构件。GTK 函数(例如 gtk_signal_connect)是用来关联 GUI 事件和对象的，而 GNOME 函数(例如 gnome_app_create_menus)则可以创建菜单。在 GNOME 程序中，需要包含一个叫作 gnome_init 的初始化函数，并将该函数置于程序的开头。要为应用程序创建一个窗口，利用函数 gnome_app_new。

下面的代码说明了函数 gnome_init 和 gnome_app_new 的使用。函数 gnome_init 的参数包括程序开始时用户输入的初始参数、一个应用程序 ID 号和版本号。用户的初始参数由特殊变量 argc 和 argv 处理。函数 gnome_app_new 的参数包括在应用程序窗口显示的标题、应用程序对象的名字。它返回新对象的地址，在本例中，该地址被赋给指针 app。

```
GtkWidget *app;
gnome_init(" ", "0.1", argc, argv);
app=gnome_app_new("Hello world", "Hello App");
```

其他操作，例如显示构件和开始交互界面等，由 GTK 函数处理。函数 gtk_widget_show_all 将显示一个构件以及它包含的任何其他构件。函数 gtk_main 将开始交互式操作，检测 GUI 事件例如鼠标单击和键按下等，并运行与这些事件关联的函数。

```
gtk_widget_show_all(app);
gtk_main();
```

下面的例子是一个简单的 GNOME 应用程序，我们仍然以“Hello World!”作为例子。

例 10-6　GNOME 的“Hello World!”程序，这个程序创建一个简单的窗口，窗口中有一个按钮，并在终端窗口的输出中显示一条信息。当单击关闭按钮后，窗口关闭。

```
1    /*ex6.c*/
2    #include <gnome.h>
3    #include <stdio.h>
4    
5    void hello(GtkWidget *widget, gpointer data)
6    {
7         g_print("Hello World!\n");
8    }
9    
10   gint deleteevent(GtkWidget *widget, GdkEvent *event, gpointer data)
11   {
12        gtk_main_quit();
13   }
14   int main(int argc, char *argv[])
15   {
16        GtkWidget *app;
17        GtkWidget *mybutton;
18        gnome_init(" ", "0.1", argc, argv);
19        mybutton=gtk_button_new_with_label("Click me");
20        app=gnome_app_new("Hello world", "Hello App");
21        gnome_app_set_contents(GNOME_APP(app),mybutton);
22        gtk_signal_connect(GTK_OBJECT(app),"delete_event",
                         GTK_SIGNAL_FUNC(deleteevent), NULL);
23        gtk_signal_connect(GTK_OBJECT(mybutton),"clicked", GTK_SIGNAL_FUNC(hello),
                         NULL);
24        gtk_widget_show_all(app);
25        gtk_main();
26   
27        return 0;
28   }
```

说明：GNOME 程序必须包含头文件 gnome.h(第 2 行)。上面的程序首先定义了 2 个回

调函数：hello 和 deleteevent。hello 输出一个简单文本“Hello World！”。deleteevent 调用 gtk_main_quit 函数结束程序(第 5~13 行)。在主函数中，定义了 2 个 GtkWidget 指针：app 和 mybutton。app 是指向主应用窗口的指针，而 mybutton 是指向一个简单按钮对象的指针(第 16~17 行)。随后利用 gnome_init 函数初始化 GNOME 界面(第 18 行)。接着利用 gtk_button_new_with_label 函数创建按钮对象，并将它的地址赋给指针 mybutton，按钮显示时带有标签“Click me”(第 19 行)。下面是利用 gnome_app_new 函数创建应用程序窗口构件(第 20 行)。然后利用 gnome_app_set_contents 将按钮放入应用程序窗口(第 21 行)。利用 gtk_signal_connect 将应用程序和 delete_event 信号关联，当用户单击关闭按钮时会发出该信号。设置它运行 closeprog 函数，而 closeprog 函数将会调用 gtk_main_quit 结束程序运行。利用 gtk_signal_connect 将按钮与鼠标单击关联，设置它运行 hellomessage 函数。只要用户一单击按钮，标准输出就会显示“Hello World”(第 22~23 行)。利用 gtk_widget_show_all 函数显示应用程序窗口及其包含的按钮，最后利用 gtk_main 开始交互界面(第 24~25 行)。

编译 GNOME 应用程序时需要调用大量的库，带着所有这些列出的库和标记的编译命令将会非常复杂以至于难以组织。与 GTK+编程类似，GNOME 提供了 gnome-config 脚本，可以在编译程序时作为参数调用这个脚本，而不必手工列出那些 GNOME 的标记，而—libs 选项生成所需的 GNOME 库列表。我们需要指定要用的库，例如 gnomeui 和 gnome，如下所示：

```
`gnome-config --cflags –libs gnome gnomeui`
```

例 10-7　程序的完整编译命令如下：

```
gcc -o ex6 ex6.c `gnome-config --cflags --libs gnome gnomeui`
```

程序在 Ubuntu 下的结果如图 10-7 所示，单击“点击我”按钮时，在终端输出窗口会显示“Hello World！”。

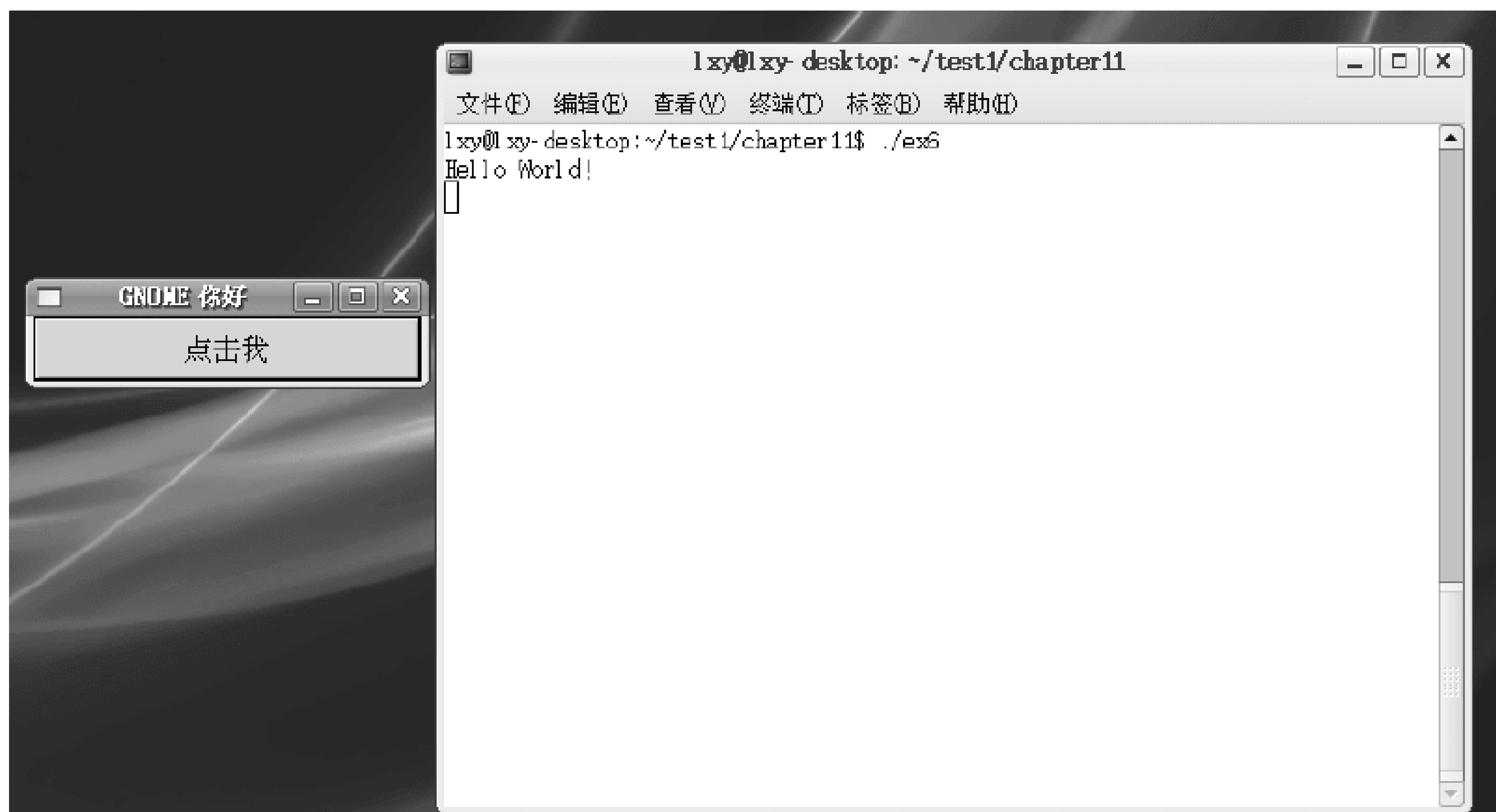


图 10-7　程序 6 的输出结果

10.3.3.1 GNOME App、工具栏和菜单构件

GnomeApp 构件是 GNOME 应用程序的基本构件。这个构件是包含菜单、工具栏和数据的主要窗口。利用 gnome_app_new 函数创建一个新的 GnomeApp 构件。这个函数以应用程序名作为它的参数。

GNOME 使我们能够为 GnomeApp 构件创建出菜单和工具条来，而它们又都可以在窗口里被最小化或从最小化状态还原。需要我们做的只是把必要的信息填写到一个数组里，然后再调用 gnome_app_create_menu 或 gnome_app_create_toolbar 函数就可以了。

菜单或工具条里通常都会有不止一个的数据项，每一个这样的数据项其属性(类型、字符串、回调函数指针、快捷键等)都要在相应的菜单数组或工具条数组里用一个结构来进行定义。这个结构的细节请查阅 liggnomeui 的 API 参考手册。在大多数情况下，菜单项都是非常简单的，因此通常可以通过 GNOME 提供的一组宏定义来简化这个结构的创建工作。每一种常见的菜单和工具条选项都有一个对应的宏定义。下面就是它们的简单介绍。

首先是一些用来创建顶层菜单的顶层宏定义，如表 10-5 所示。

表 10-5 顶层菜单宏定义

菜　单	宏　定　义
File	GNOMEUIINFO_MENU_FILE_TREE(tree)
Edit	GNOMEUIINFO_MENU_EDIT_TREE(tree)
View	GNOMEUIINFO_MENU_VIEW_TREE(tree)
Settings	GNOMEUIINFO_MENU_SETTINGS_TREE(tree)
Windows	GNOMEUIINFO_MENU_WINDOWS_TREE(tree)
Help	GNOMEUIINFO_MENU_HELP_TREE(tree)
Game	GNOMEUIINFO_MENU_GAME_TREE(tree)

在顶层菜单里面又定义了超过三十个用来创建常用菜单项的宏定义。这些宏定义可以给每个菜单项加上小图标和快捷键。只需要定义一个将会在该菜单项被选中时调用执行的回调函数和一个传递给那个函数的指针[格式为“(cb, data)”]就行了。一些常用的宏定义如表 10-6 所示。

表 10-6 菜单项的常用宏定义

顶层菜单	菜　单　项	宏　定　义
File	New	GNOMEUIINFO_MENU_NEW_ITEM(label, hint, cb, data)
	Open	GNOMEUIINFO_MENU_OPEN_ITEM(cb, data)
	Save	GNOMEUIINFO_MENU_SAVE_ITEM(cb, data)
	Print	GNOMEUIINFO_MENU_PRINT_ITEM(cb, data)
	Exit	GNOMEUIINFO_MENU_EXIT_ITEM(cb, data)

(续表)

顶层菜单	菜 单 项	宏 定 义
Edit	Cut	GNOMEUIINFO_MENU_CUT_ITEM(cb, data)
	Copy	GNOMEUIINFO_MENU_COPY_ITEM(cb, data)
	Paste	GNOMEUIINFO_MENU_PASTE_ITEM(cb, data)
Settings	Preferences	GNOMEUIINFO_MENU_PREFERENCES_ITEM(cb, data)
Help	About	GNOMEUIINFO_MENU_ABOUT_ITEM(cb, data)

工具条的情况与菜单是很相似的，我们要用 GNOMEUIINFO_ITEM_STOCK (label,tooltip, callback, stock_id)宏定义来创建一个相应的数组，其中，stock_id 是打算用为该项目图标的一个预定义图标的 id 值。这些预定义图标的完整清单也列在 1ibgnomeui 的参考手册里。

另外还有几个特殊的宏定义，包括 GNOMENUIINFO_SEPERATOR——它的作用是创建一条用来物理性地分隔菜单项或工具条项目的线条；GNOMEUIINFO_ITEM(label, tooltip, cb)-——它的作用是添加一个带有用户自己的标签的菜单项。GNOMEUIINFO_END—它的作用是表示数组到此结束，等等。下面我们看一个例子。

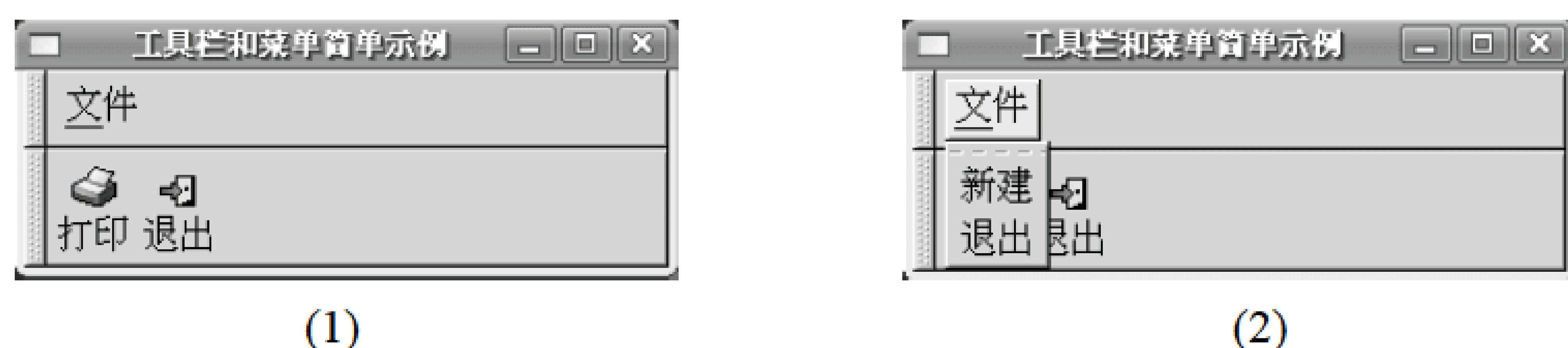
例 10-8　GNOME 中的菜单和工具条。

```
1     /*ex7.c*/
2     #include <gnome.h>
3     #include <stdio.h>
4
5     void callback(GtkWidget *widget, gpointer data)
6     {
7             g_print("Item Selected\n");
8     }
9
10    GnomeUIInfo file_menu[]={
11            GNOMEUIINFO_ITEM_NONE("新建" ,"This is the menubar info", callback),
12    GNOMEUIINFO_MENU_EXIT_ITEM(gtk_main_quit, NULL),
13            GNOMEUIINFO_END
14    };
15
16    GnomeUIInfo menubar[]= {
17    GNOMEUIINFO_MENU_FILE_TREE(file_menu),
18            GNOMEUIINFO_END
19    };
20
21    GnomeUIInfo toolbar[]= {
22            GNOMEUIINFO_ITEM_STOCK("打印", "This is another tooltip", callback,
```

```
GNOME_STOCK_PIXMAP_PRINT),
23          GNOMEUIINFO_ITEM_STOCK("退出", "Exit the application", gtk_main_quit,
                                  GNOME_STOCK_PIXMAP_EXIT),
24          GNOMEUIINFO_END
25      };
26
27      int main(int argc, char *argv[])
28      {
29          GtkWidget     *app;
30          GtkWidget *button;
31          GtkWidget *label;
32
33          gnome_init("example", "0.1", argc, argv);
34          app=gnome_app_new("example", "工具栏和菜单简单例子");
35
36          gtk_signal_connect(GTK_OBJECT(app), "delete_event",
                            GTK_SIGNAL_FUNC(gtk_main_quit), NULL);
37
38          gnome_app_create_menus(GNOME_APP(app),menubar);
39          gnome_app_create_toolbar(GNOME_APP(app),toolbar);
40
41          gtk_widget_show_all(app);
42          gtk_main();
43
44          return 0;
45      }
```

说明：程序首先创建一个回调函数，当有项目被选中时，它会向终端输出相应的文字(第 5~8 行)。然后创建一个有 2 个元素的数组，它们将被放在 File 菜单里。这 2 个元素一个是用来调用回调函数的选项，另外一个是退出选项(第 10~14 行)。接下来创建菜单的结构，它只有一个顶层的文件菜单，它指向刚才创建的数组(第 16~19 行)。工具条的情况与菜单类似。我们创建了一个有 2 个元素的数组，它们一个是打印按钮，一个是退出按钮(第 21~25 行)。最后，创建出菜单和工具条并把它们摆放到窗口里去(第 38~39 行)。

程序在 Ubuntu 下的运行结果如图 10-8 所示。其中图(2)是单击 FILE 菜单后弹出的子菜单的情形。



(1) (2)

图 10-8 程序 7 的输出结果

以上我们简要介绍了用GTK+编写GNOME程序的基本框架、GNOME App构件、工具栏和菜单构件的使用方法，GNOME的构件还有很多，如对话框、图标、画布等，很明显没有足够的篇幅来讨论所有的构件，而且毫无疑问我们只是刚接触到GNOME的皮毛。读者可以自行到gnome的网站上查阅相关的资料以及它们的使用方法等。

10.4 小　　结

这一章介绍了Linux下GUI编程的一些基本概念，包括X窗口系统，使用Xlib编写X窗口系统应用程序的方法，随后介绍了X工具包，重点讲解了基础性的开发工具包GTK+，学习了用GTK+编写X应用程序的基本方法。最后介绍了用GTK+编写GNOME程序的方法。介绍了GNOME App、工具栏和菜单构件的简单使用方法。学习完本章后，读者应对Linux下的GUI编程方法有初步的了解。

习　　题

一、填空题

1. X窗口系统主要由______、______、______、______组成。

2. 用Xlib编程时，客户程序连接和解除连接一个X服务器时要使用______和______函数。

3. 比较知名的X工具包包括______、______、______、______、______。

GNOME库提供了GNOME应用软件中使用的最高级函数。在它们的下面是______库。而______库又是由GIMP工具箱______和GIMP绘图工具箱______库组成。

4. 利用gcc编译器和GTK+库来编译GTK+程序。要在命令行中指定GTK+库，可以使用______脚本命令。

5. 要创建一个简单的GNOME程序，首先为GNOME构件定义______对象，然后利用______来初始化程序和定义用户的构件。

二、选择题

1. GTK+是GNOME应用软件使用的构件集，它的外观和感觉最初是来源于______。

(A) Xt　(B) Motif　(C) Openlook　(D) Qt

2. 创建一个新的GTK窗口结构的函数是______。

(A)gtk_init　(B) gtk_window_new　(C) gtk_widget_show　(D) gtk_main

3. ex.c 是一个 GTK 程序，下列编译命令正确的是：

(A) gcc –o ex ex.c gtk-config –cflags --libs

(B) gcc –o ex ex.c `gtk-config –cflags –libs`

(C) gcc –o ex ex.c ‘gtk-config –cflags –libs

(D) gcc –o ex ex.c “gtk-config –cflags –libs”

4. 当把鼠标指针放置在按钮上时，按下鼠标键，产生的 GTK 按钮信号是______。

(A) pressed (B) released (C) clicked (D) enter

5. GNOME 程序的初始化函数为______。

(A) gnome_init (B) gnome_app_new (C) gtk_init (D) gtk_main

三、上机题

1. 编写一个 Xlib 程序，在窗口中输出“How are you？”。

2. 在例中，当单击关闭按钮时，程序并不退出。改写程序，在单击关闭按钮时，使程序退出，同时在终端窗口输出“Program Exit！”

3. 编写一个 GTK 程序，窗口包含一个标签，一个输入框和一个按钮，标签和按钮的名字自己定义。

4. 编写一个包含菜单构件的 GNOME 程序。顶层菜单包含“文件”和“编辑”两项，其中，“文件”中又包含“新建”、“打开”、“关闭”菜单项。“编辑”包含“剪切”、“复制”、“粘贴”菜单项。

CHAPTER 11

飞机票网络售票系统

本书前面的章节中详细介绍了 Linux 环境下 C 语言编程的基本方法，相信读者阅读之后会对 Linux 下的 C 语言编程有一个基本的认识。为了便于讲解 Linux 下 C 语言编程的基本方法，本书前面章节中的例题都很简短，为了使读者进一步加深了解，本章给出综合实例，其中使用了前面几乎所有章节的知识。希望读者看完例子后，也能亲自上机实验一下。

我们将设计并实现一个飞机票网络售票系统的模拟程序。首先说明系统的总体设计，主要包含通信报文格式的设计、服务器端的设计和客户端的设计。在服务器端使用多线程的编程技术，使用一个服务线程来为一个客户的连接服务。

下面我们将对程序的源码进行详细的解析。希望通过这个示例使读者对 Linux 下的 C 程序的编写有更进一步的认识和提高。(需要申明的是，本章的程序还仅仅是演示程序，限于篇幅，还有一些细节没有充分地考虑和处理) 。

11.1　系 统 框 架

限于篇幅，只能对网络售票系统进行简单的示意，所以模拟系统完成的功能比较简单。假设在系统中有一台服务器和其他一些客户计算机。服务器和这些客户计算机通过 Internet 相连。

服务器通常是处理能力较强、稳定性较好的计算机。在服务器上存放着飞机票的相关信息，包括飞机航班、对应的价格和剩余数量等，这些数据通常存在数据库中。

系统中设有许多的售票网点，每个售票网点拥有一台或多台客户计算机(售票终端)。售票员在独立的售票终端上工作，这些售票终端将和服务器进行通信。整个系统示意图如图 11-1 所示。

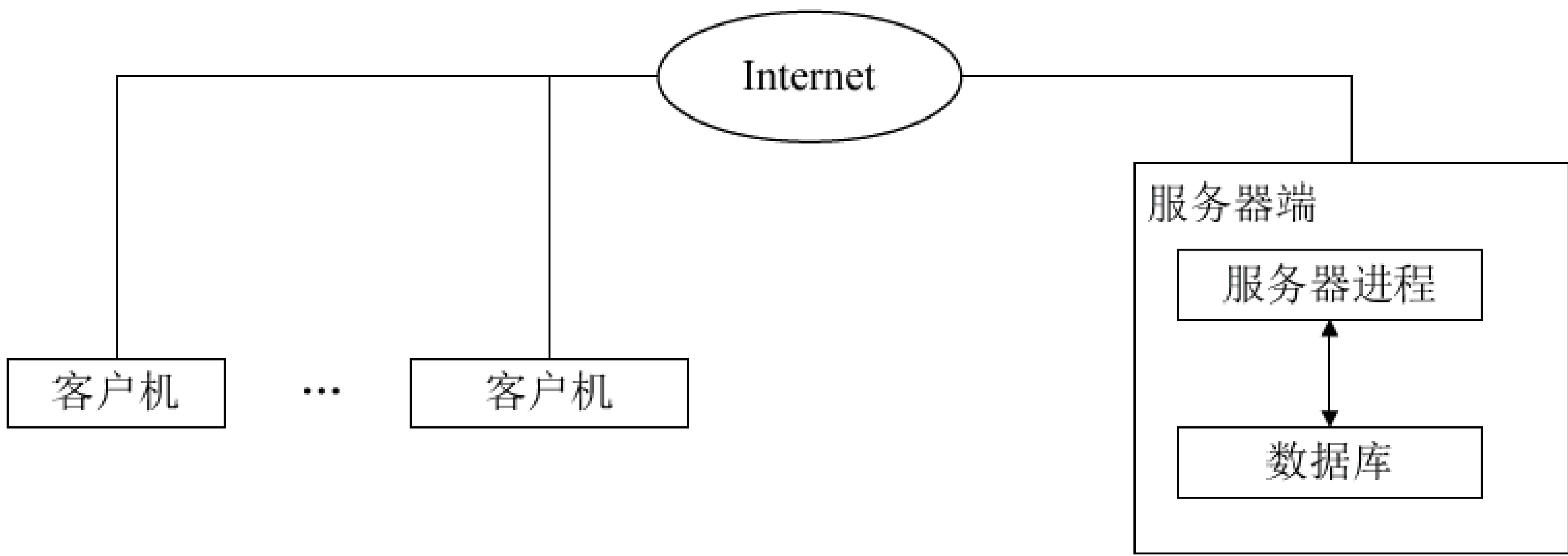


图 11-1　系统结构图

每个售票员在每次售票中，需要输入购买的航班号和购买的数量。如果系统中该航班的机票剩余张数大于或等于需要购买的张数，则购买成功。如果剩余票数不足，则提示剩余票数，本次购买失败。

此外，售票员还可以在售票终端查询航班的剩余机票信息，可以查询单次航班，也可以查询所有航班。

系统必须保证售票的安全性。当多个终端同时购买同一航班的机票时，系统不应当存在异常。

11.1.1　数据格式

根据说明的功能，这里使用的数据记录的格式比较简单。客户端发送请求的数据报文由 4 部分组成：请求消息类型、购买的航班号和购买的票数。次序和字节数参见图 11-2 所示。

4 字节	4 字节	4 字节	4 字节
消息类型	航班号	购买的票数	票价

图 11-2　数据记录格式

其中，消息 leix 的长度为 4 字节，它用于说明客户端请求服务的类型，定义的请求服务类型包括，购买机票、查询特定航班机票、查询所有航班机票。在客户端的请求消息类型中票价为空，具体的数值由服务器来填充。

服务器端返回的回应报文结构同客户端一样，不过消息类型含义不一样。

如果客户端请求是购买机票，当指定航班的剩余票数小于请求的票数，则回应记录中将填写该次剩余的票数和机票单价，同时消息类型为购买失败；如果指定航班的剩余票数大于或等于请求的票数，则填写购得的本次航班的票数和机票价钱总和，同时消息类型为购买成功。客户端的程序可以根据这个回应报文判断是否购买成功。

如果客户端的请求是查询航班，则回应记录中将填写查询航班的剩余票数和机票单价，同时消息类型为查询成功。

如果服务器收到的请求代码有误，则服务器在回应报文中的消息类型填写为未知代码，客户端根据此代码可知请求在发送过程中出错。

在此只是对网络购票系统进行模拟，所以定义的数据结构的格式很简单，如果要开始一个真正实用的网络购票系统或其他网络购物系统，则要根据实际情况，进行更复杂的数据格式定义。

11.1.2 服务器端程序框架

服务器端使用多线程技术。主线程将使用倾听套接字，并接收新连接，当新连接接入后，主线程将为新连接创建一个服务线程，并为新的服务线程分配一个线程缓冲区，将服务线程使用需要的信息保存在这个线程缓冲区中。

新的服务线程被创建后，它将使用新连接接入后的连接套接字，来接收客户进程的请求报文，并在处理后，发送回应报文。

下面分别说明主线程和服务线程对于网络套接字 socket、线程缓冲区 thread_buffer 和信号的使用情况以及服务器端的工作过程等。

11.1.2.1 网络套接字 socket 的使用

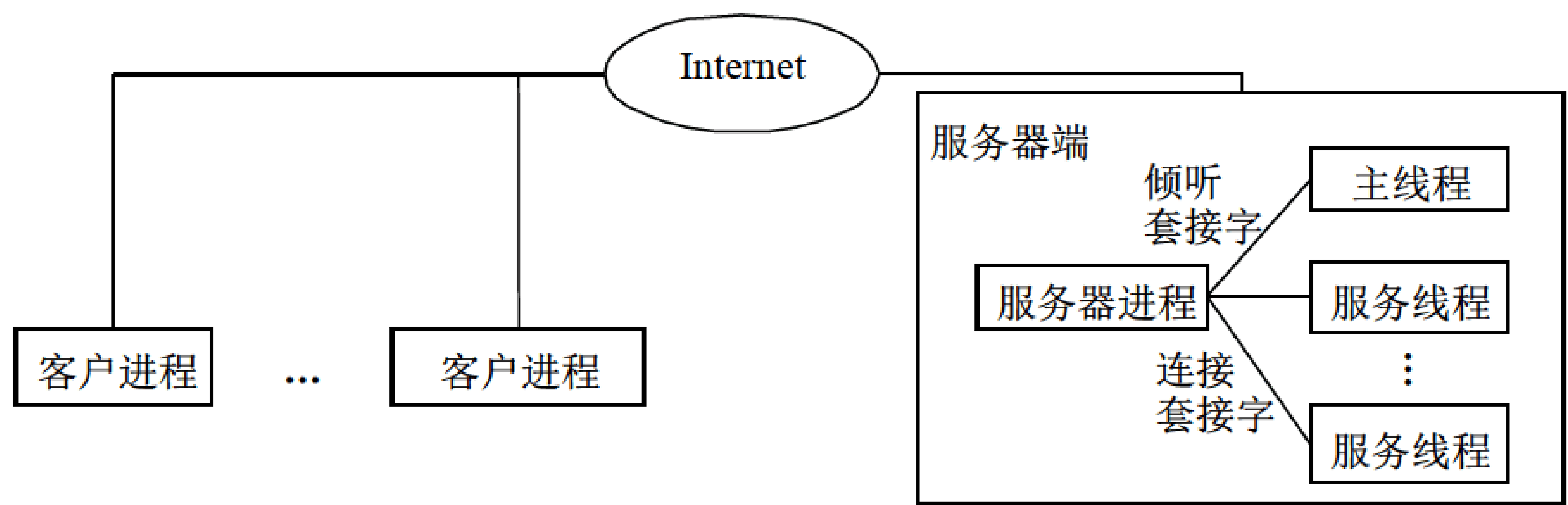


图 11-3 网络套接字的使用

在图 11-3 中显示了主线程和其他服务线程对网络套接字的使用情况。主线程监视侦听套接字，而服务线程使用连接套接字。

在多线程并发的环境中，侦听套接字和连接套接字是在同一个进程空间中，所以主线程在接收新连接并创建服务线程后，不应当关闭这个连接套接字，如果关闭了，则访问它的服务线程将出错。主线程以后不再访问它，而只有对应的服务线程去访问这个连接套接字。

11.1.2.2 线程缓冲区结构体 struct thread_buffer

```
typedef thread_buffer_struct_t {
    /*保存对应线程的线程号*/
    int         tid;
    /*保存对应的客户机的 IP 地址*/
    unsigned long ip_addr;
    /*该线程使用的连接套接字描述符*/
    int         conn_fd;
    /*线程缓冲区的状态*/
```

```
    int         buffer_status;
} thread_buffer_struct;
```

在服务器程序中，线程缓冲区是重要的数据结构。各个变量的含义已经在注释中说明了。

主线程管理和分配 thread_buffer 结构数组，一个服务线程对应一个 thread_buffer，当主线程分配线程缓冲区时，需要检测 buffer_status 变量的值，而服务线程在退出前，需要将它使用的线程缓冲区释放。所谓释放就是需要修改 buffer_status 变量的值，所以主线程和服务线程间需要对 buffer_status 变量进行互斥，我们可以为每一个 buffer_status 变量设置一个互斥锁，但是这样需要较多的锁资源。这里使用了一个互斥锁来对结构数组中的所有 buffer_status 变量进行互斥保护，这样对并发性有一定的影响，但是，由于线程退出的操作并不是经常发生的，所以不会对并发性有很大的影响。

11.1.2.3 服务器端工作流程

服务进程的主线程将在侦听套接字上接收 TCP 的新连接，当主线程返回一个新的连接套接字描述符后，它将首先检测客户端的 IP 地址是否与在线程缓冲区中保存的 IP 地址相同，如果有相同的，说明客户进程被重新启动了，此时主线程将关闭那个服务线程，服务线程退出，而后主线程将线程缓冲区释放。

如果没有发现相同的 IP 地址，说明的确是新的客户连接，主线程将为这个连接分配新的线程缓冲区，如果没有空闲的缓冲区则关闭这个连接。如果分配线程缓冲区成功，则将服务线程需要使用的参数保存在线程结构中，而后创建一个新的服务线程，并将分配的缓冲区的序号传递给服务线程。而后主线程继续接收新连接。

服务线程通过缓冲区序号获取套接字描述符的值，而后使用这个套接字描述符同客户端进行通信，并同其他的服务线程一起进行售票的请求处理。

当服务线程发现对端进程关闭连接时或者发生其他严重错误时，它将释放它占用的线程缓冲区，而后退出。

11.1.3 客户端程序框架

在客户端设置一个售票进程，售票员可以在多个终端上启动多个终端进程(每个终端只能启动一个终端进程)，这些终端进程将接收数据并显示处理结果，它们通过网络与服务器进行通信，完成所有数据的发送和接收。

下面简要说明客户端与服务器通信用到的消息结构和客户端工作流程。

11.1.3.1 消息结构

消息结构的代码如下所示：

```
/*客户端与服务器端使用的消息类型定义*/
#define     INITIAL_VALUE      65535
```

```
/*客户端使用的消息代码含义*/
#define     DISCONNECT         0
#define     BUY_TICKET    1
#define     INQUIRE_ONE  2
#define     INQUIRE_ALL  3

/*服务器端使用的消息代码含义*/
#define     BUY_SUCCEED              255
#define     BUY_FAILED               256
#define     INQUIRE_SUCCEED 257
#define     UNKNOWN_CODE           258

/*服务器与客户端使用的消息结构定义*/
struct stMessage {
    //消息类型。客户端可以取值为 DISCONNECT：断开连接；BUY_TICKET：购买机票；
      INQUIRE_ONE：查询特定航班机票；INQUIRE_ALL：查询所有航班机票
    unsigned int   msg_type;
    //航班号
    unsigned int   flight_ID;
    //机票张数
    unsigned int   ticket_num;
    //机票价钱
    unsigned int   ticket_total_price;
} message;
```

客户端与服务器将使用 stMessage 结构进行信息的传递。其中 msg_type 是消息的标识，它不是消息的内容，flight_ID 是航班号，ticket_num 是票数，ticket_total_price 是票价，由服务器端进行填充。

11.1.3.2　客户端工作流程

售票员可以在不同的终端启动终端进程。终端进程获取售票员的输入，然后将信息组织成消息结构，发送给服务器，并等待服务器的响应，如果服务器进程在指定的时间内没有发送回处理结果，则客户端进程认为该 TCP 连接已经不可用，它将给出错误提示信息。如果服务器端在指定的时间内，发送回应信息，它将对结果进行处理，并显示处理的结果。

关于客户瑞和服务器端详细的工作流程，将在程序源码中进一步解析。

11.2　程序源代码和说明

程序源代码分为服务器端源代码和客户端源代码，我们分别进行说明。

11.2.1　服务器端源代码

服务器端源代码分为 server.c、servicethread.h、globalapi.h、icket.h、threadbuff.h 等文件构成。各个文件的功能和主要内容如下：

server.c：服务器端主线程的实现代码。

servicethread.h：服务器端服务线程的实现代码，其中，service_thread 函数是服务线程的执行函数。

globalapi.h：定义了需要的一些函数库的头文件、客户端与服务器端通信使用的消息结构以及一些相关的函数。

ticket.h：定义结构 tickt_struct 以及结构的访问函数。

threadbuff.h：定义结构 thread_buff 以及结构的访问函数。

下面给出各个文件的源代码。

11.2.1.1　global.h

global.h 的代码如下所示：

```
1	#ifndef	__GLOBALAPI_H
2	#define	__GLOBALAPI_H
3
4	#include <sys/types.h> /*基本的系统数据类型*/
5	#include <sys/socket.h> /*基本的套接字的定义*/
6	#include <sys/time.h>
7	#include <netinet/in.h>
8	#include <arpa/inet.h>
9	#include <netdb.h>
10	#include <stdio.h>
11	#include <stdlib.h>
12	#include <sys/stat.h>
13	#include <unistd.h>
14	#include <sys/un.h>
15	#include <string.h>
16	#include <pthread.h>
17	#include <gnome.h>
18
19	/*服务器端使用的端口*/
20	#define	SERVER_PORT_NO	8889
21
22	/*客户端与服务器端使用的消息类型定义*/
23	#define	INITIAL_VALUE	65535
24
```

```
25    /*客户端使用的消息代码含义*/
26    #define     DISCONNECT          0
27    #define     BUY_TICKET   1
28    #define     INQUIRE_ONE  2
29    #define     INQUIRE_ALL  3
30
31    /*服务器端使用的消息代码含义*/
32    #define     BUY_SUCCEED              255
33    #define     BUY_FAILED               256
34    #define     INQUIRE_SUCCEED 257
35    #define     UNKNOWN_CODE             258
36
37    /*服务器与客户端使用的消息结构定义*/
38    struct stMessage {
39            //消息类型。客户端可以取值为 DISCONNECT：断开连接；BUY_TICKET：购买机
              票；IINQUIRE_ONE：查询特定航班机票；   INQUIRE_ALL：查询所有航班机票
40            unsigned int        msg_type;
41            //航班号
42            unsigned int        flight_ID;
43            //机票张数
44            unsigned int        ticket_num;
45            //机票价钱
46            unsigned int        ticket_total_price;
47    } message;
48
49    void init_message()
50    {
51            message.msg_type=INITIAL_VALUE;
52            message.flight_ID=0;
53            message.ticket_num=0;
54            message.ticket_total_price=0;
55    }
56
57    /*服务器端的线程缓冲区的最大数量*/
58    #define     THREAD_BUFF_NUM 128
59
60    /*提示信息输出*/
61    #define INFO_NUM    10
62    #define     INFO_OCCUPIED     1
63    #define     INFO_FREED    0
64
65    struct info_t {
66            int            status;            /*INFO_OCCUPIED or INFO_FREED*/
```

```
67          char  msg[512];          /*contents of message*/
68      }   info[INFO_NUM];
69
70      pthread_mutex_t          info_mutex;
71      /*初始化界面输出信息缓冲区*/
72      void init_info()
73      {
74          int i;
75          for (i=INFO_NUM;i>0;i--)
76              info[i-1].status=INFO_FREED;
77              sprintf(info[i-1].msg," ");
78      }
79
80      /*分配一个空闲的界面输出信息缓冲区，如果没有空闲的缓冲区则返回－1*/
81      int get_free_info()
82      {
83          int i,ret;
84          /*注意对互斥锁的操作，这些操作必须是成对的(加锁和解锁)，否则会发生死锁的
            情况*/
85          pthread_mutex_lock(&info_mutex);
86          for(i=0;i<INFO_NUM; i++)
87              if(info[i].status==INFO_FREED) {
88                  ret=i;
89                  pthread_mutex_unlock(&info_mutex);
90                  break;
91              }
92          if(i==INFO_NUM) {
93              ret=-1;
94              pthread_mutex_unlock(&info_mutex);
95          }
96          return ret;
97      }
98
99      /*释放信息缓冲区，对 info_status 的访问同样需要使用互斥保护*/
100     void free_info(int index)
101     {
102         int i;
103         pthread_mutex_lock(&info_mutex);
104         if(info[index].status==INFO_OCCUPIED)
105             info[index].status=INFO_FREED;
106         pthread_mutex_unlock(&info_mutex);
107     }
108
```

```
109    void add_info(char *src)
110    {      int i;
111           while((i=get_free_info())==-1)
112           {
113                  sleep(1);
114           }
115
116           /*添加信息*/
117           pthread_mutex_lock(&info_mutex);
118           info[i].status=INFO_OCCUPIED;
119           strcpy(info[i].msg, src);
120           pthread_mutex_unlock(&info_mutex);
121    }
122
123    #endif
```

说明：globallib.h 定义了需要的一些函数库的头文件以及一些系统常数。其中SERV_PORT_NO(第 20 行)定义了服务器端使用的连接端口号。第 23~47 行定义了客户端与服务器端通信使用的消息格式，消息格式的含义前面已经解释，这里不再说明。第 49~55 行是在通信前对消息进行初始化的函数。第 57 行服务器端的线程缓冲区的最大数量，这个数值也说明了服务器可同时服务的客户端的数量。第 61~121 行定义了在界面输出相关信息的函数和变量。由于程序是多线程的，分为界面线程、监听线程以及服务线程。监听线程与服务线程要输出相关信息需要通过界面线程。为此定义了界面线程与监听线程和服务线程交互使用的信息缓冲区；缓冲区的内容是监听线程与服务线程要输出的字符串信息。在界面线程中，定义了一个定时器回调函数，周期性地检查缓冲区是否有要输出的信息，如果有需要输出的信息，则在界面显示。由于界面线程、监听线程和服务线程都要访问信息缓冲区，为了防止产生冲突，定义了访问信息缓冲区的互斥锁(第 70 行)。init_info 函数(第72~78 行)用来初始化界面输出信息缓冲区。get_free_info 函数(第 81~97 行)分配一个空闲的输出信息缓冲区。free_info 函数(第 100~107 行)是释放信息缓冲区。add_info 函数(第 109~121 行)则用来往信息缓冲区中添加要显示的文本信息。

11.2.1.2 threadbuff.h

threadbuff.h 的代码如下所示：

```
1      /*threadbuff.h*/
2      #ifndef     __THREAD_BUFF_H
3      #define     __THREAD_BUFF_H
4
5      #include "globalapi.h"
6
7      /*定义线程缓冲区的使用状态*/
```

```
8       #define             BUFF_OCCUPIED     1
9       #define             BUFF_FREED        0
10
11      /*线程缓冲区结构*/
12
13      typedef struct thread_buff_struct_t {
14              /*线程缓冲区的索引号*/
15              int buff_index;
16              /*保存对应线程的线程号*/
17              int     tid;
18              /*保存对应的客户机的 IP 地址*/
19              unsigned long ip_addr;
20              /*该线程使用的连接套接字描述符*/
21              int     conn_fd;
22              /*线程缓冲区的状态*/
23              int     buff_status;
24      } thread_buff_struct;
25
26      thread_buff_struct thread_buff[THREAD_BUFF_NUM];
27      /*  用于线程缓冲区互斥使用的互斥锁*/
28      /*由于当主线程分配线程缓冲区时需要检测 buff_status 变量的值，而服务线程在退出前，
        需要将它使用的线程缓冲区释放。所谓释放就是需要修改 buff_status 变量的值，所以主线
        程和服务线程间需要对 buff_status 进行互斥，可以为每一个 buff_status 变量设置一个互斥
        锁，但这样需要较多的系统资源。这里只使用了一个互斥锁来对结构数组中的所有
        buff_status 变量进行互斥保护*/
29      pthread_mutex_t buff_mutex;
30
31      /*初始化线程缓冲区*/
32      void init_thread_buff()
33      {
34              int index;
35              for(index=0; index<THREAD_BUFF_NUM;index++) {
36                      thread_buff[index].tid=-1;
37                      thread_buff[index].buff_status=BUFF_FREED;
38              }
39      }
40
41      /*分配一个空闲的线程缓冲区，如果没有空闲的缓冲区则返回－1*/
42      int get_free_buff()
43      {
44              int i,ret;
45              /*注意对互斥锁的操作，这些操作必须是成对的(加锁和解锁)，否则会发生死锁的情况*/
46              pthread_mutex_lock(&buff_mutex);
```

```
47          for(i=0;i<THREAD_BUFF_NUM; i++)
48                  if(thread_buff[i].buff_status==BUFF_FREED) {
49                          ret=i;
50                          pthread_mutex_unlock(&buff_mutex);
51                          break;
52                  }
53          if(i==THREAD_BUFF_NUM) {
54                  ret=-1;
55                  pthread_mutex_unlock(&buff_mutex);
56          }
57          return ret;
58  }
59
60  /*释放线程缓冲区，对 buff_status 的访问同样需要使用互斥保护*/
61  void free_buff(int index)
62  {
63          pthread_mutex_lock(&buff_mutex);
64          if(thread_buff[index].buff_status==BUFF_OCCUPIED)
65                  thread_buff[index].buff_status=BUFF_FREED;
66          pthread_mutex_unlock(&buff_mutex);
67  }
68
69  /*检查线程缓冲区中是否有重复连接，因为可能客户端的通信进程终止后重新启动，此时
应当终止原来它所对应的服务线程，再重新创建一个服务线程，并为这个新的服务线程分
配线程缓冲区*/
70  void check_connection(unsigned long ip_addr)
71  {
72          int i,j;
73          struct in_addr in;
74          char msg[512];
75          /*检查所有的线程缓冲区*/
76          pthread_mutex_lock(&buff_mutex);
77          for(i=0;i<THREAD_BUFF_NUM;i++) {
78                  /*发现重复连接*/
79                  if((thread_buff[i].buff_status!=BUFF_FREED)&&thread_buff[i].ip_addr==ip_addr) {
80                          in.s_addr=htonl(ip_addr);
81                          sprintf(msg,"重复连接：%s，旧连接将关闭！\n",inet_ntoa(in));
82                          add_info(msg);
83                          pthread_cancel(thread_buff[i].tid);
84
85                          pthread_join(thread_buff[i].tid,NULL);
86                          /*退出的线程不释放它的缓冲区，释放工作由主线程来处理*/
87                          thread_buff[i].tid=0;
```

```
88                    thread_buff[i].buff_status=BUFF_FREED;
89                    close(thread_buff[i].conn_fd);
90                }
91            }
92            pthread_mutex_unlock(&buff_mutex);
93    }
94
95    #endif
```

说明：文件的第 8~9 行定义了线程缓冲区的使用状态。第 13~24 行定义了线程缓冲区结构。第 29 行定义了用于线程缓冲区使用的互斥锁。由于当监听线程分配线程缓冲区时需要检测 buff_status 变量的值，而服务线程在退出前，需要将它使用的线程缓冲区释放，所谓释放就是需要修改 buff_status 变量的值，所以监听线程和服务线程间需要对 buff_status 进行互斥，可以为每一个 buff_status 变量设置一个互斥锁，但这样需要较多的系统资源。这里只使用了一个互斥锁来对结构数组中的所有 buff_status 变量进行互斥保护。第 32~67 行定义了几个与线程缓冲区操作相关的函数。包括初始化线程缓冲区函数 init_thread_buffer(第 32~39 行)，分配空闲线程缓冲区函数 get_free_buffer(第 42~58 行)，释放缓冲区函数 free_buff(第 61~67 行)。第 70~93 行定义了 check_connection 函数，它用来检查线程缓冲区中是否有重复连接。文件的其他说明参照程序注释。

11.2.1.3　ticket.h

ticket.h 的代码如下所示：

```
1     /*ticket.h*/
2     #ifndef     __TICKET_H
3     #define     __TICKET_H
4     #include "globalapi.h"
5
6     #define     FLIGHT_NUM 10           //航班总数
7
8     /*机票的一个简单描述，flight_ID 表示航班号，ticket_num 表示机票剩余票数*/
9     typedof struct ticket_struct_t {
10            int         flight_ID;
11            int         ticket_num;
12            int         ticket_price;     //票价
13            /*多个线程操作时，必须对机票的剩余数量进行保护。由于这样的操作比较频繁，
              所以应当对每一个 ticket_num 使用不同的互斥锁，否则将对线程间并行性有较大影响*/
14            pthread_mutex_t ticket_mutex;
15    } ticket_struct;
16    ticket_struct ticket_list[FLIGHT_NUM];
17
18    /*init_ticket_list:初始化 ticket_list 数组*/
```

```
19    void init_ticket_list()
20    {
21        int i;
22        for(i=0; i<FLIGHT_NUM;i++) {
23            ticket_list[i].flight_ID=i+1;
24            ticket_list[i].ticket_num=100;
25            ticket_list[i].ticket_price=300*(i+1);
26        }
27    }
28
29
30    #endiff
```

说明：ticket.h 定义结构 ticket_struct(第 9~15 行)以及结构的初始化函数(第 19~27 行)。ticket_struct 结构是对机票的简单描述，包括航班号、剩余的机票数量、票价。通常，这些数据应该存放在数据库中，服务器程序要通过访问数据库获取这些数据。由于我们在此只是对网络售票系统进行简单模拟，所以在程序中直接生成相关数据，这并不影响程序的功能。读者如有兴趣，可以在此程序的基础上进行完善。init_ticket_list 函数(第 18~25 行)简单地填写票数、航班号和票价，完成 ticket_list 数组的初始化。函数的具体说明参照程序中的注释。

11.2.1.4 servicethread.h

servicethread.h 的代码如下所示：

```
1     *servicethread.h*/
2     #ifndef    __SERVICE_THREAD_H
3     #define    __SERVICE_THREAD_H
4
5     #include "ticket.h"
6     #include "threadbuff.h"
7
8     /*thread_err：服务线程的错误处理函数，由于服务器端使用的是多线程技术，服务线程发
      生错误时，不能像在多进程的情况下，简单地调用 exit()终止进程。在多线程下，服务线
      程必须将使用的资源释放后，调用 pthread_exit()退出，并且在需要进行线程间同步的情况
      下，还需要做一些线程同步的工作，才能退出。这个特点在多线程编程中是非常重要的*/
9     static void thread_err(char *s, int index)
10    {
11        int i;
12        char msg[512];
13        /*获取空闲的界面输出信息缓冲区，如果没有空闲的,延迟一段时间后继续获取*/
14        sprintf(msg,"线程 %d 发生致命错误：,%s\n",(unsigned short)pthread_self(),s);
15        add_info(msg);
16        //info_print(strmsg,serverwindow);
17        /*释放线程使用的线程缓冲区*/
```

```
18          free_buff(index);
19          pthread_exit(NULL);
20      }
21
22      /*service_thread:服务线程的线程函数。服务线程根据函数的参数中获取自身使用的线程缓
        冲区的序号，而后根据这个序号从线程缓冲区中获取需要的参数*/
23      void * service_thread(void *p)
24      {
25          int conn_fd;
26          int buff_index;
27          char  send_buf[1024],recv_buf[512];
28          int             ret,i,cnt;
29          uint16_t     nbyte;
30          struct sockaddr_in       peer_name;
31          int             peer_name_len;
32          unsigned int        required_ticket_num;
33          int             pos;
34          thread_buff_struct *pstruct;
35          char msg[512];
36          /*获取线程使用的线程缓冲区的序号*/
37          pstruct=(thread_buff_struct *)p;
38          buff_index=pstruct->buff_index;
39          pstruct->tid=pthread_self();
40
41          /*从线程缓冲区中获取通信使用的套接字描述符*/
42          conn_fd=pstruct->conn_fd;
43
44          /*打印远端主机地址*/
45          peer_name_len=sizeof(peer_name);
46          ret=getpeername(conn_fd,(struct sockaddr*)&peer_name, &peer_name_len);
47          if(ret==-1)
48              thread_err("获取远端主机地址出错",buff_index);
49
50          sprintf(msg,"新连接-->线程 ID：%d，连接 ID：%d，线程缓冲区索引号：%d，远端地
            址：%s，端口号：%d\n",(unsigned short)pstruct->tid,conn_fd, buff_index,
            inet_ntoa(peer_name.sin_addr), ntohs(peer_name.sin_port));
51          add_info(msg);
52          while(1) {
53              /*从网络中获取数据记录*/
54              ret=recv(conn_fd,recv_buf,sizeof(message),0);
55              /*接收出错*/
56              if(ret==-1) {
57                  sprintf(msg,"线程：%d  在连接：%d  接收出错。连接将关闭。\n",(unsigned
```

```
                    short)pstruct->tid, conn_fd);
58                  add_info(msg);
59                  thread_err(msg, buff_index);
60              }
61              /*ret==0 说明客户端连接已关闭*/
62              if(ret==0) {
63                  sprintf(msg,"线程  %d  的连接( ID：%d ) 客户端已关闭。服务器端连接
                    也将关闭。\n",(unsigned short)pstruct->tid, conn_fd);
64                  add_info(msg);
65                  close(conn_fd);
66                  free_buff(buff_index);
67                  pthread_exit(NULL);
68              }
69
70              /*ret 为其他值说明接收到了客户端的请求消息*/
71              init_message();
72              memcpy(&message,recv_buf,sizeof(message));
73              switch(message.msg_type) {
74                  case  DISCONNECT:
75                      sprintf(msg,"线程 %d  的连接(ID：  %d ) 客户端已关闭。服务器端
                        连接也将关闭。\n",(unsigned short)pstruct->tid, conn_fd);
76                      add_info(msg);
77                      close(conn_fd);
78                      free_buff(buff_index);
79                      pthread_exit(NULL);
80                      break;
81                  case  BUY_TICKET :
82                      for(i=0; i<FLIGHT_NUM; i++) {
83                          pthread_mutex_lock(&ticket_list[i].ticket_mutex);
84                          if(ticket_list[i].flight_ID==message.flight_ID) {
85                              if(ticket_list[i].ticket_num>=message.ticket_num) {        //
                                剩余票数大于请求票数
86                                  message.msg_type=BUY_SUCCEED;
87                                  message.ticket_total_price=message.
                                    ticket_num*ticket_list[i].ticket_price;
88                                  ticket_list[i].ticket_num-=message.ticket_num;
89                                  pthread_mutex_unlock(&ticket_list[i].ticket_mutex);
90                                  sprintf(msg,"购买成功！航班号：%d, 票数：%d, 总
                                    票价：%d\n",message.flight_ID, message.ticket_num,
                                    message.ticket_total_price);
91                                  add_info(msg);
92                                  memcpy(send_buf,&message,sizeof(message));
93                                  ret=send(conn_fd, send_buf, sizeof(message), 0);
```

```
94                              if(ret<0)
95                                  thread_err("发送数据出错\n", buff_index);
96                              break;
97                          } else {
        //剩余票数不足，购买失败
98                              message.msg_type=BUY_FAILED;
99                              required_ticket_num=message.ticket_num;
100                             message.ticket_num=ticket_list[i].ticket_num;
101                             pthread_mutex_unlock(&ticket_list[i].ticket_mutex);
102                             sprintf(msg,"购买失败！航班号：%d，剩余票数：
                                 %d，请求票数：%d\n",message.flight_ID,
                                message.ticket_num,required_ticket_num);
103                             add_info(msg);
104                             memcpy(send_buf,&message,sizeof(message));
105                             ret=send(conn_fd, send_buf, sizeof(message), 0);
106                             if(ret<0)
107                                 thread_err("发送数据出错\n", buff_index);
108                             break;
109                         }
110                     }
111                 pthread_mutex_unlock(&ticket_list[i].ticket_mutex);
112             }
113             break;
114             case  INQUIRE_ONE:
115                 for(i=0; i<FLIGHT_NUM; i++) {
116                     pthread_mutex_lock(&ticket_list[i].ticket_mutex);
117                     if(ticket_list[i].flight_ID==message.flight_ID) {
118                         message.msg_type=INQUIRE_SUCCEED;
119                         message.ticket_num=ticket_list[i].ticket_num;
120                          message.ticket_total_price=
                        ticket_list[i].ticket_price;
121                         pthread_mutex_unlock(&
                        ticket_list[i].ticket_mutex);
122                         sprintf(msg,"查询成功！航班号：%d，票数：
                        %d，票价：%d\n",message.flight_ID,
                        message.ticket_num, message.ticket_total_price);
123                         add_info(msg);
124                         memcpy(send_buf,&message,sizeof(message));
125                         ret=send(conn_fd, send_buf, sizeof(message), 0);
126                         if(ret<0)
127                             thread_err("发送数据出错\n", buff_index);
128                         break;
129                     }
```

```
130                         pthread_mutex_unlock(&ticket_list[i].ticket_mutex);
131                     }
132                     break;
133                 case  INQUIRE_ALL:
134                     pos=0;
135 
136                     for(i=0; i<FLIGHT_NUM; i++) {
137                         pthread_mutex_lock(&ticket_list[i].ticket_mutex);
138                         message.msg_type=INQUIRE_SUCCEED;
139                         message.flight_ID=ticket_list[i].flight_ID;
140                         message.ticket_num=ticket_list[i].ticket_num;
141                         message.ticket_total_price=ticket_list[i].ticket_price;
142                         pthread_mutex_unlock(&ticket_list[i].ticket_mutex);
143                         if(i==0) {
144                             sprintf(msg,"查询所有航班号成功！ \n");
145                             add_info(msg);
146                         }
147                         sprintf(msg,"航班号：%d, 票数：%d, 票价：
                            %d\n",message.flight_ID, message.ticket_num,
                            message.ticket_total_price);
148                         add_info(msg);
149                         memcpy(send_buf+pos,&message,sizeof(message));
150                         pos+=sizeof(message);
151                     }
152                     ret=send(conn_fd, send_buf, pos, 0);
153 
154                     if(ret<0)
155                         thread_err("发送数据出错\n", buff_index);
156                     break;
157                 default     :
158                     message.msg_type=UNKNOWN_CODE;
159                     memcpy(send_buf, &message,sizeof(message));
160                     ret=send(conn_fd, send_buf, sizeof(message), 0);
161                     if(ret<0)
162                         thread_err("发送数据出错\n", buff_index);
163             }
164         }
165     }
166 
167     #endif
```

说明：servicethread.h 定义了服务器端服务线程的有关函数。thread_err(第 9~20 行)是服务线程的错误处理函数，由于服务器端使用的是多线程技术，服务线程发生错误时，不

能像在多进程的情况下，简单地调用 exit 函数终止进程。在多线程下，服务线程必须将使用的资源释放后，调用 pthread_exit()退出，并且在需要进行线程间同步的情况下，还需要做一些线程同步的工作，才能退出。这个特点在多线程编程中是非常重要的。service_thread(第 23~165 行)是服务线程的线程函数。服务线程根据函数的参数获取自身使用的线程缓冲区的序号，而后根据这个序号从线程缓冲区中获取需要的参数(第 37~42 行)。随后，服务线程就等待接收从客户端发来的消息，根据消息类型的不同，分别进行相应的处理，处理完毕后，发送处理结果给客户端，再次等待客户端的请求(第 52~165 行)。关于处理的详细过程可参考程序中的注释。

11.2.1.5 server.c

server.c 的代码如下所示：

```
1    /*server.c*/
2
3    #include "servicethread.h"
4
5    int listen_fd, conn_fd;                           //监听 socket，连接 socket
6    struct sockaddr_in server, cli_addr;                  //服务器地址信息，客户端地址信息
7    int ret, buffer_index, i,cli_len;
8    unsigned long ip_addr;
9    int flag=1;
10   pthread_t listentid,servicetid;                          //监听线程 ID,服务线程 ID
11
12   static GtkWidget        *app;                            /*程序主窗口*/
13   static GtkWidget *frame, *vbox, *box2 ,*table;      /*box2 用来封装文本构件与垂直滚动条*/
14   static GtkWidget *serverwindow , *vscrollbar;//界面信息输出窗口，用来输出相关提示信息
15
16   /*监听线程*/
17   void * listen_thread(void *p)
18   {
19       char msg[512];
20       while(1) {
21           /*接受远端的 TCP 连接请求*/
22           cli_len=sizeof(cli_addr);
23           conn_fd=accept(listen_fd,(struct sockaddr *)&cli_addr, &cli_len);
24           if(conn_fd<0)
25               continue;
26
27           ip_addr=ntohl(cli_addr.sin_addr.s_addr);
28           /*检测重复连接*/
29           check_connection(ip_addr);
30           /*分配线程缓冲区*/
```

```
31                  buffer_index=get_free_buff();
32                  if(buffer_index<0) {
33                          sprintf(msg,"没用空闲的线程缓冲区。\n");
34                          add_info(msg);
35                          close(conn_fd);
36                          continue;
37                  }
38                  /*填写服务线程需要的信息*/
39                  pthread_mutex_lock(&buff_mutex);
40                  thread_buff[buffer_index].buff_index=buffer_index;
41                  thread_buff[buffer_index].ip_addr=ip_addr;
42                  thread_buff[buffer_index].conn_fd=conn_fd;
43                  thread_buff[buffer_index].buff_status=BUFF_OCCUPIED;
44                  pthread_mutex_unlock(&buff_mutex);
45
46                  /*创建新的服务线程*/
47                  ret=pthread_create(&servicetid, NULL, service_thread,
                        &thread_buff[buffer_index]);
48                  if(ret==-1) {
49                          sprintf(msg,"创建服务线程出错！\n");
50                          add_info(msg);
51                          close(conn_fd);
52                          /*释放线程缓冲区*/
53                          pthread_mutex_lock(&buff_mutex);
54                          thread_buff[buffer_index].tid=0;
55                          thread_buff[buffer_index].buff_status=BUFF_FREED;
56                          pthread_mutex_unlock(&buff_mutex);
57                  }
58          }
59  }
60
61  int isserveropened=FALSE;          //服务器端是否开启标志位
62
63  void startserver(GtkWidget *widget, gpointer data);
64  void stopserver(GtkWidget *widget, gpointer data);
65  void inquireone();
66  void inquireall();
67  void displaycontents(GtkWidget *widget, gpointer data);
68  void about(GtkWidget *widget, gpointer data);
69
70  /*生成服务器操作菜单项*/
71  GnomeUIInfo server_operation_menu[]={
72          GNOMEUIINFO_ITEM_NONE("开启服务器" ,"开启服务器", startserver),
```

```
73          GNOMEUIINFO_ITEM_NONE("关闭服务器" ,"关闭服务器", stopserver),
74          GNOMEUIINFO_ITEM_NONE("退出" ,"退出程序", gtk_main_quit),
75          GNOMEUIINFO_END
76      };
77
78      /*生成航班查询菜单项*/
79      GnomeUIInfo inquire_menu[]={
80          GNOMEUIINFO_ITEM_NONE("特定航班查询" ,"查询某一特定航班机票信息",
                inquireone),
81          GNOMEUIINFO_ITEM_NONE("所有航班查询" ,"查询所有航班机票信息",
                inquireall),
82          GNOMEUIINFO_END
83      };
84
85      /*生成帮助菜单项*/
86      GnomeUIInfo help_menu[]={
87          GNOMEUIINFO_ITEM_NONE("显示内容" ,"显示帮助内容", displaycontents),
88          GNOMEUIINFO_ITEM_NONE("关于" ,"关于此程序说明", about),
89          GNOMEUIINFO_END
90      };
91
92      /*生成顶层菜单项*/
93      GnomeUIInfo menubar[]= {
94          GNOMEUIINFO_SUBTREE("服务器操作(_S)", server_operation_menu),
95          GNOMEUIINFO_SUBTREE("机票查询(_Q)", inquire_menu),
96          GNOMEUIINFO_SUBTREE("帮助(_H)", help_menu),
97          GNOMEUIINFO_END
98      };
99
100     GnomeUIInfo toolbar[]= {
101         GNOMEUIINFO_ITEM_STOCK("开启服务器","开启服务器", startserver,
              GNOME_STOCK_PIXMAP_PRINT),
102         GNOMEUIINFO_ITEM_STOCK("关闭服务器","关闭服务器", stopserver,
            GNOME_STOCK_PIXMAP_PREFERENCES),
103         GNOMEUIINFO_ITEM_STOCK("特定航班查询","查询某一特定航班机票信息",
            inquireone, GNOME_STOCK_PIXMAP_SAVE),
104         GNOMEUIINFO_ITEM_STOCK("所有航班查询","查询所有航班机票信息",
            inquireall, GNOME_STOCK_PIXMAP_SAVE),
105         GNOMEUIINFO_ITEM_STOCK("退出", "退出程序", gtk_main_quit,
            GNOME_STOCK_PIXMAP_EXIT),
106         GNOMEUIINFO_END
107     };
108
```

```
109  /*消息内容输出函数*/
110  void display_info(char *msg, GtkWidget *window)
111  {
112          gtk_text_freeze (GTK_TEXT (window));
113          gtk_text_insert (GTK_TEXT (window), NULL, &window->style->black, NULL,msg, -1);
114          gtk_text_thaw (GTK_TEXT (window));
115  }
116
117  /*开启服务器操作*/
118  void startserver(GtkWidget *widget, gpointer data)
119  {
120          char msg[512];          //提示信息
121          GtkWidget *isenable;
122
123          /*初始化数据结构*/
124          init_thread_buff();
125          init_ticket_list();
126
127          if(!isserveropened)
128          {
129                  /*创建套接字*/
130                  listen_fd=socket(AF_INET,SOCK_STREAM,0);
131                  if(listen_fd<0) {
132                          sprintf(msg,"创建监听套接字出错！  \n");
133                          display_info(msg,serverwindow);
134                          return;
135                  }
136
137                  /*填写服务器的地址信息*/
138                  server.sin_family=AF_INET;
139                  server.sin_addr.s_addr=htonl(INADDR_ANY);
140                  server.sin_port=htons(SERVER_PORT_NO);
141
142                  setsockopt(listen_fd,SOL_SOCKET,SO_REUSEADDR,(void *)&flag,sizeof(int));
143
144                  /*绑定端口*/
145                  ret=bind(listen_fd,(struct sockaddr*)&server, sizeof(server));
146                  if(ret<0) {
147                          sprintf(msg,"绑定 TCP 端口：%d  出错！  \n",SERVER_PORT_NO);
148                          display_info(msg,serverwindow);
149                          close(listen_fd);
150                          return;
151                  }
```

```
152
153                /*转化成倾听套接字*/
154                listen(listen_fd,5);
155                ret=pthread_create(&listentid, NULL, listen_thread, NULL);
156                if(ret==-1) {
157                        sprintf(msg,"创建监听线程出错！  \n");
158                        display_info(msg,serverwindow);
159                        close(listen_fd);
160                        return;
161                }
162
163                //成功后输出提示信息
164                sprintf(msg,"服务器端开启成功！服务器在端口：%d 准备接受连接！
                    \n",SERVER_PORT_NO);
165                display_info(msg,serverwindow);
166                isserveropened=TRUE;
167
168                /*开启服务器菜单项和工具条灰化*/
169                isenable=toolbar[0].widget;
170                gtk_widget_set_sensitive(isenable,FALSE);
171                isenable=server_operation_menu[0].widget;
172                gtk_widget_set_sensitive(isenable,FALSE);
173
174                /*关闭服务器菜单项和工具条使能*/
175                isenable=toolbar[1].widget;
176                gtk_widget_set_sensitive(isenable,TRUE);
177                isenable=server_operation_menu[1].widget;
178                gtk_widget_set_sensitive(isenable,TRUE);
179
180        }
181    }
182
183    /*关闭服务器操作*/
184    void stopserver(GtkWidget *widget, gpointer data)
185    {
186        GtkWidget *isenable;
187        char msg[512];
188        if(isserveropened)
189        {
190
191                pthread_mutex_lock(&buff_mutex);
192                for(i=0;i<THREAD_BUFF_NUM;i++) {
193                        if(thread_buff[i].buff_status!=BUFF_FREED) {
```

```
194                         /*退出服务线程*/
195                         pthread_cancel(thread_buff[i].tid);
196                         pthread_join(thread_buff[i].tid,NULL);
197                         /*退出的线程不释放它的缓冲区，释放工作由主线程来处理*/
198                         thread_buff[i].tid=0;
199                         thread_buff[i].buff_status=BUFF_FREED;
200                         close(thread_buff[i].conn_fd);
201                     }
202                 }
203 
204             pthread_mutex_unlock(&buff_mutex);
205             pthread_cancel(listentid);
206             pthread_join(listentid,NULL);
207             close(listen_fd);
208             sprintf(msg,"服务器端成功关闭！  \n");
209             display_info(msg,serverwindow);
210             isserveropened=FALSE;
211 
212             /*开启服务器菜单项和工具条使能*/
213             isenable=toolbar[0].widget;
214             gtk_widget_set_sensitive(isenable,TRUE);
215             isenable=server_operation_menu[0].widget;
216             gtk_widget_set_sensitive(isenable,TRUE);
217 
218             /*关闭服务器菜单项和工具条灰化*/
219             isenable=toolbar[1].widget;
220             gtk_widget_set_sensitive(isenable,FALSE);
221             isenable=server_operation_menu[1].widget;
222             gtk_widget_set_sensitive(isenable,FALSE);
223         }
224 }
225 /*查询特定航班对话框的“确定”、“取消”按钮的回调函数*/
226 int button_inquireone(GnomeDialog *dialog, gint id,gpointer data)
227 {
228         GtkWidget *flight=data;
229         GtkWidget *mbox;
230         char msg[512];
231         int     flight_ID;
232         if(id==0) {                     //如果“确定”按钮被单击
233             sprintf(msg,gtk_entry_get_text(GTK_ENTRY(flight)));
234             flight_ID=atoi(msg);
235             if(flight_ID<=0 || flight_ID>10) {  //判断输入的航班号是否正确，不正确的话，
                                               给出提示信息，重新输入
```

```
236                         mbox = gnome_message_box_new ("输入的航班号错误！请重新输入！
                                ",GNOME_MESSAGE_BOX_INFO,GNOME
                                _STOCK_BUTTON_OK,NULL ) ;
237                         gtk_widget_show (mbox);
238                         gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
239                         gnome_dialog_set_parent(GNOME_DIALOG
                            ( mbox),GTK_WINDOW(dialog));
240                         gtk_entry_set_text(GTK_ENTRY(flight)," ");
241                         return ;
242                 }
243                 for(i=0;i<FLIGHT_NUM;i++) {
244                         pthread_mutex_lock(&ticket_list[i].ticket_mutex);
245                         if(ticket_list[i].flight_ID==flight_ID) {
246                                 sprintf(msg,"你查询的航班号是：%d，剩余票数：%d,票价：
                                %d\n",ticket_list[i].flight_ID,ticket_list[i].
                                ticket_num,ticket_list[i].ticket_price);
247                                 display_info(msg,serverwindow);
248                                 pthread_mutex_unlock(&ticket_list[i].ticket_mutex);
249                                 break;
250                         }
251                         pthread_mutex_unlock(&ticket_list[i].ticket_mutex);
252                 }
253                 gnome_dialog_close(dialog);
254         }
255         else
256                 gnome_dialog_close(dialog);
257 }
258 
259 /*查询某一特定航班*/
260 void inquireone()
261 {
262         GtkWidget *dialog;
263         GtkWidget *label,*flight_entry;
264         dialog = gnome_dialog_new(_("查询特定航班机票信息")
                    ,GNOME_STOCK_BUTTON_OK ,GNOME_
                    STOCK_BUTTON_CANCEL ,NULL) ;
265 
266         label=gtk_label_new("请输入要查询的航班号(1-10)：");
267         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),label,TRUE,TRUE ,0 ) ;
268         gtk_widget_show (label);
269 
270         flight_entry=gtk_entry_new();
271         gtk_entry_set_visibility(GTK_ENTRY(flight_entry),TRUE);
```

```
272         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),flight_entry,
                                FALSE, FALSE,0);
273         gtk_widget_show (flight_entry);
274
275         gnome_dialog_set_default(GNOME_DIALOG(dialog),0);      //设定确定作为缺省按钮
276
277         gtk_signal_connect(GTK_OBJECT(dialog),"clicked" ,GTK_SIGNAL_FUNC(button_inq
                                uireone) ,flight_entry) ; //设定“确定”“取消”按钮的回调函数
278         gtk_window_set_modal (GTK_WINDOW (dialog), TRUE);
279         gtk_widget_show (dialog);
280         gnome_dialog_set_parent(GNOME_DIALOG( dialog),GTK_WINDOW(app));   //设定
                              对话框的父窗口为主程序窗口
281     }
282
283     void inquireall()
284     {
285
286         int i;
287         char msg[512];
288         for(i=0;i<FLIGHT_NUM;i++) {
289             sprintf(msg,"航班号：%d，剩余票数：%d，票价：
                %d\n",ticket_list[i].flight_ID,ticket_list[i].ticket_num, ticket_list[i].ticket_price);
290             display_info(msg,serverwindow);
291         }
292     }
293
294     /*帮助对话框和关于对话框的确定按钮处理函数*/
295     void dialog_ok(GnomeDialog *dialog, gint id,gpointer data)
296     {
297         gnome_dialog_close(dialog);
298     }
299
300     /*帮助———显示内容*/
301     void displaycontents(GtkWidget *widget, gpointer data)
302     {
303         GtkWidget *dialog;
304         GtkWidget *label;
305         char msg[4][512]={"开启服务器：开启服务器，准备接受客户端连接","关闭服务器：
                              关闭服务器端","特定航班查询：查询某一特定航班机票信息","所
                              有航班查询：查询所有航班机票信息"};
306         dialog = gnome_dialog_new(_("帮助"),_ ( "确定" ) ,NULL ,NULL) ;
307         for(i=0;i<4;i++) {
308             label=gtk_label_new(msg[i]);
```

```
309                 gtk_box_pack_start(GTK_BOX(GNOME_DIALOG
                    (dialog)->vbox),label,TRUE,TRUE ,0 ) ;
310                 gtk_widget_show (label);
311         }
312
313         gtk_signal_connect(GTK_OBJECT(dialog),"clicked" ,GTK_SIGNAL_FUNC(dialog_ok)
                        ,&dialog) ;
314         gtk_window_set_modal (GTK_WINDOW (dialog), TRUE);
315         gtk_widget_show (dialog);
316         gnome_dialog_set_parent(GNOME_DIALOG( dialog),GTK_WINDOW(app));
317 }
318
319
320 /*帮助———关于*/
321 void about(GtkWidget *widget, gpointer data)
322 {
323         GtkWidget *dialog;
324         GtkWidget *contentlabel,*versionlabel,*authorlabel,*copyrightlabel;
325         dialog = gnome_dialog_new(_("关于本程序"),_ ( "确定" ) ,NULL ,NULL) ;
326         contentlabel=gtk_label_new(_("网络售票模拟系统服务器端"));
327         copyrightlabel=gtk_label_new(_("Copyright 2009"));
328         authorlabel=gtk_label_new(_("lxy"));
329         versionlabel=gtk_label_new(_("版本：0.1"));
330         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),contentlabel,TRUE,
                    TRUE ,0 ) ;
331         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),versionlabel,TRUE,
                    TRUE ,0 ) ;
332         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),copyrightlabel,TRU
                    E,TRUE ,0 ) ;
333         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),authorlabel,TRUE,
                    TRUE ,0 ) ;
334
335         gtk_widget_show (contentlabel);
336         gtk_widget_show (versionlabel);
337         gtk_widget_show (copyrightlabel);
338         gtk_widget_show (authorlabel);
339
340
341         gtk_signal_connect(GTK_OBJECT(dialog),"clicked" ,GTK_SIGNAL_FUNC(dialog_ok)
            ,&dialog) ;
342         gtk_window_set_modal (GTK_WINDOW (dialog), TRUE);
343         gtk_widget_show (dialog);
344         gnome_dialog_set_parent(GNOME_DIALOG( dialog),GTK_WINDOW(app));
```

```
345 }
346
347 /*定时器回调函数，从界面输出信息缓冲区中取出要显示的信息输出到屏幕上*/
348 gint info_print()
349 {
350         int i;
351         pthread_mutex_lock(&info_mutex);
352         for(i=0;i<INFO_NUM;i++)
353                 if(info[i].status==INFO_OCCUPIED) {
354                         display_info(info[i].msg,serverwindow);
355                         info[i].status=INFO_FREED;
356                 }
357         pthread_mutex_unlock(&info_mutex);
358         return TRUE;
359 }
360
361 int main(int argc, char *argv[])
362 {
363         GtkWidget       *isenable;
364
365         init_ticket_list();
366         //界面初始化部分
367         gtk_set_locale();
368         gnome_init("example", "0.1", argc, argv);
369         app=gnome_app_new("example", "网络售票模拟系统服务器端");
370         gtk_signal_connect(GTK_OBJECT(app), "delete_event",
        GTK_SIGNAL_FUNC(gtk_main_quit), NULL);
371         /*创建菜单和工具栏*/
372         gnome_app_create_menus(GNOME_APP(app),menubar);
373         gnome_app_create_toolbar(GNOME_APP(app),toolbar);
374
375         /*启动程序时使关闭服务器工具条和菜单项灰化*/
376         if(!isserveropened) {
377                 isenable=toolbar[1].widget;
378                 gtk_widget_set_sensitive(isenable,FALSE);
379                 isenable=server_operation_menu[1].widget;
380                 gtk_widget_set_sensitive(isenable,FALSE);
381         }
382
383         gtk_window_set_default_size((GtkWindow *)app, 800, 600);
384         vbox=gtk_vbox_new(FALSE,0);
385         gnome_app_set_contents(GNOME_APP(app),vbox);
386
```

```
387         /*创建服务器输出信息窗口*/
388         frame=gtk_frame_new(NULL);
389         gtk_frame_set_label(GTK_FRAME(frame),"服务器信息");
390         gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT);
391
392         gtk_box_pack_start(GTK_BOX(vbox), frame, TRUE,TRUE,10);      /*最后一个参数控
            制间隔*/
393         gtk_widget_show(vbox);
394
395         box2 = gtk_vbox_new (FALSE, 10);
396         gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
397         gtk_container_add(GTK_CONTAINER(frame),box2);
398         gtk_widget_show (box2);
399
400         table = gtk_table_new (2, 2, FALSE);
401         gtk_table_set_row_spacing (GTK_TABLE (table), 0, 2);
402         gtk_table_set_col_spacing (GTK_TABLE (table), 0, 2);
403         gtk_box_pack_start (GTK_BOX (box2), table, TRUE, TRUE, 0);
404         gtk_widget_show (table);
405
406         serverwindow=gtk_text_new (NULL, NULL);
407         gtk_text_set_editable (GTK_TEXT (serverwindow), FALSE);   //文本信息不可编辑
408         gtk_text_set_word_wrap(GTK_TEXT(serverwindow),TRUE ); //自动换行
409         gtk_table_attach (GTK_TABLE (table), serverwindow, 0, 1, 0, 1, GTK_EXPAND |
                    GTK_SHRINK | GTK_FILL,
410         GTK_EXPAND | GTK_SHRINK | GTK_FILL, 0, 0);
411         gtk_widget_show (serverwindow);
412
413         //为输出窗口添加竖直滚动条
414         vscrollbar = gtk_vscrollbar_new (GTK_TEXT (serverwindow)->vadj);
415         gtk_table_attach (GTK_TABLE (table), vscrollbar, 1, 2, 0, 1,GTK_FILL,
                GTK_EXPAND | GTK_SHRINK | GTK_FILL, 0, 0);
416         gtk_widget_show (vscrollbar);
417         gtk_widget_realize (serverwindow);
418
419         gtk_widget_show_all(app);
420         g_timeout_add(500,info_print,NULL);   //定时更新界面输出信息
421         gtk_main();
422
423         return 0;
424     }
```

说明：server.c 是服务器端的界面主程序文件。程序第 5~14 行定义了相关的变量，包

括连接 socket、监听 socket、服务器地址信息、客户端地址信息、程序主窗口等。

第 17~59 行定义了监听线程函数。它的作用是接受客户端 TCP 连接请求，检测是否重复连接，填写服务线程需要使用的相关信息，然后建立服务线程，随后再次等候客户端的连接请求。

第 71~107 行定义了界面的菜单和工具栏，如图 11-4 所示。

第 110~359 行定义了菜单和工具栏相应的处理函数。关于函数的具体实现请参考程序注释。

第 361~425 行是服务器程序的 main 函数，在 main 函数中完成界面初始化(第 367~370 行)，生成菜单和工具栏(第 372~373 行)，并设置程序刚启动时菜单和工具栏的状态(第 376~381 行)，定义程序启动时窗口大小(第 383~385 行)，随后创建服务器信息输出窗口(第 388~417 行)，最后是显示所有窗口，定义定时器，用来定时更新界面输出信息，进入 gtk 循环(第 419~421 行)。关于程序详细说明，请参考注释。

以上就是服务器端的源代码。对服务器端的源代码进行编译连接时，由于服务器端是 GNOME 应用程序，在连接时需要指定使用 gnomeui 库，因此相应的编译命令如下所示：

```
$ gcc globalapi.h ticket.h threadbuff.h servicethread.h ticketserver.c -o server `gnome-config --cflags --libs gnomeui`
```

编译连接完成后，在当前目录下，生成了 server 的可执行文件。在终端命令行下输入下列命令即可执行：

```
$ ./server
```

执行结果如图 11-4 所示。

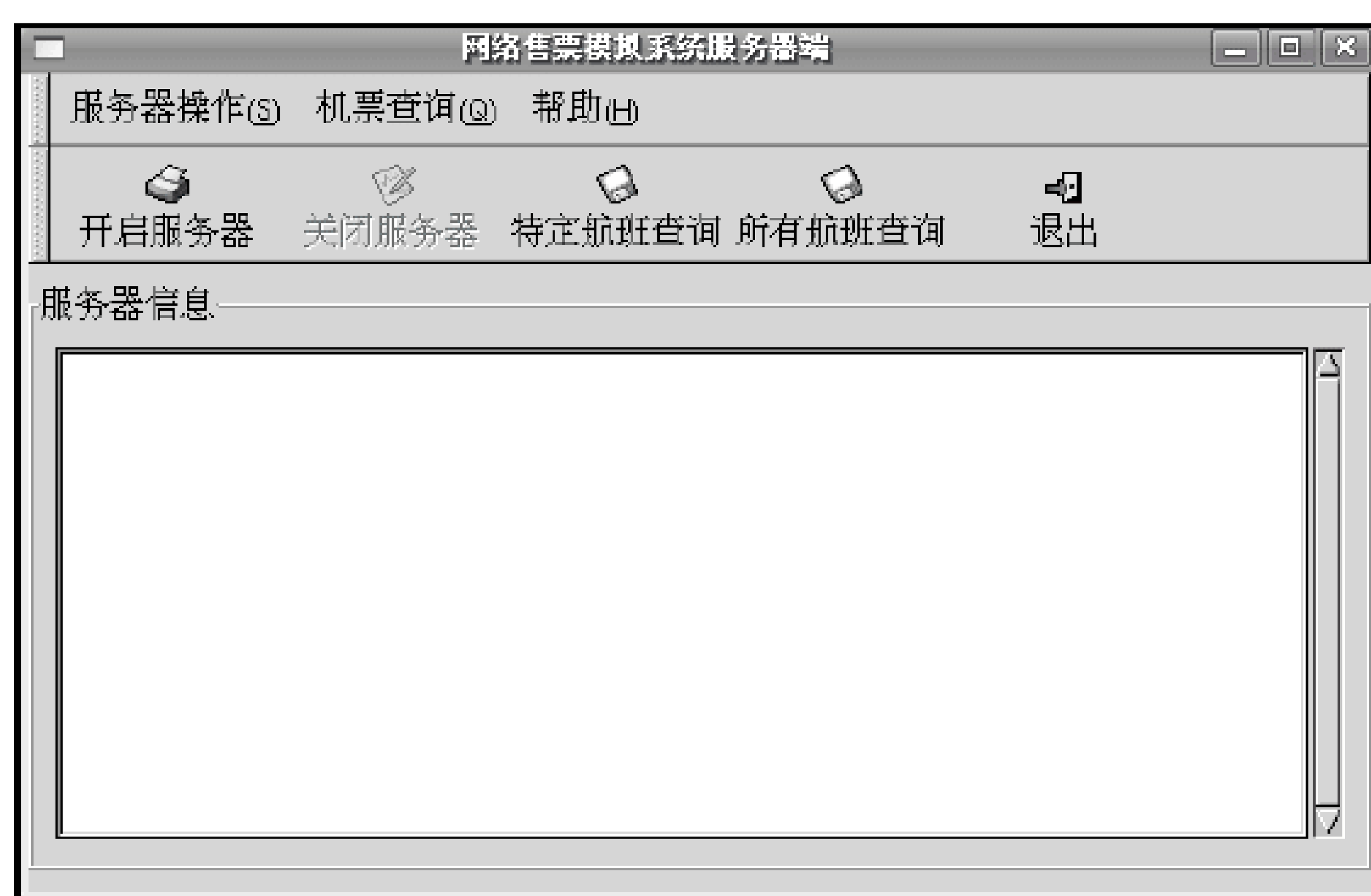


图 11-4　服务器端程序界面

程序有 3 个顶层菜单和 5 个常用工具按钮。“服务器操作”菜单包括“开启服务器”、“关闭服务器”以及“退出”3 项。它们与工具栏中的对应项功能是相同的。“机票查询”菜单包括“特定航班查询”、“所有航班查询”，它们与工具栏的对应项功能相同。以下操作以工具栏为例进行说明，这些操作也可通过选择相应的菜单项来完成，不再另行说明。“开启服务器”、“特定航班查询”、“所有航班查询”工具按钮是激活的，而“关闭服务器”

按钮是灰的。单击“开启服务器”按钮，开启成功，则界面如图 11-5 所示。

开启成功后，“开启服务器”工具按钮以及“服务器操作”菜单中的“开启服务器”项变灰，而关闭服务器及“服务器操作”菜单中的“关闭服务器”项使能。



图 11-5　服务器开启成功界面

服务器开启成功后，就可以接受客户端的连接请求。此外，服务器端也可以进行机票查询，包括查询特定航班机票和所有航班机票。当单击“特定航班查询”按钮时，系统弹出对话框如图 11-6 所示。

在此对话框中可以输入要查询的航班号。如果输入的航班号有错误，当点击“确定”时，系统会弹出错误提示，如图 11-7 所示。

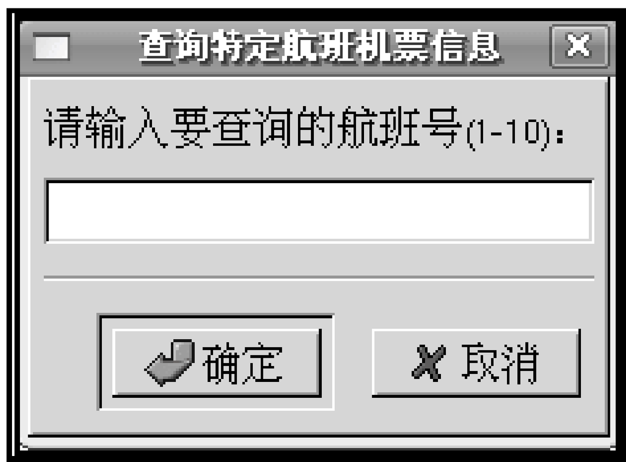


图 11-6　查询特定航班机票信息对话框

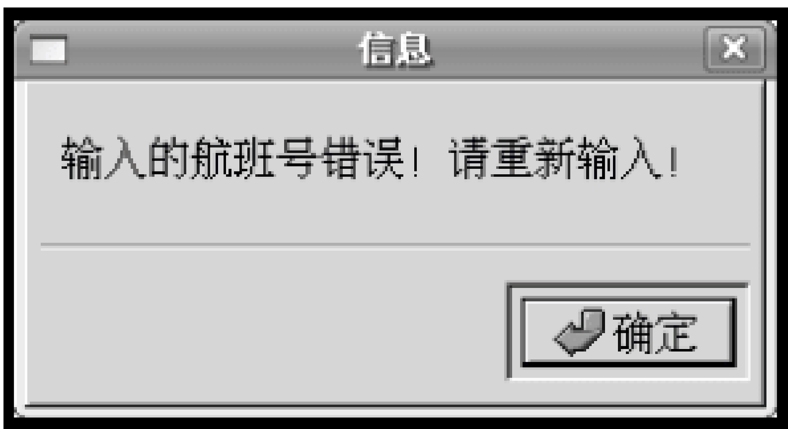


图 11-7　输入航班号错误提示对话框

当输入航班号正确时，查询对话框消失，程序输出如图 11-8 所示。



图 11-8　查询特定航班输出结果

如果要查询所有航班机票信息，则单击“所有航班查询”按钮，程序输出如图 11-9 所示。

当需要关闭服务器时，只需单击“关闭服务器”按钮，关闭后，程序输出如图 11-10 所示。此时，服务器不能接受客户端的连接请求。“关闭服务器”按钮以及“服务器操作”菜单中的“关闭服务器项”禁用，而“开启服务器”按钮和“服务器操作”菜单中的“开启服务器”项使能。

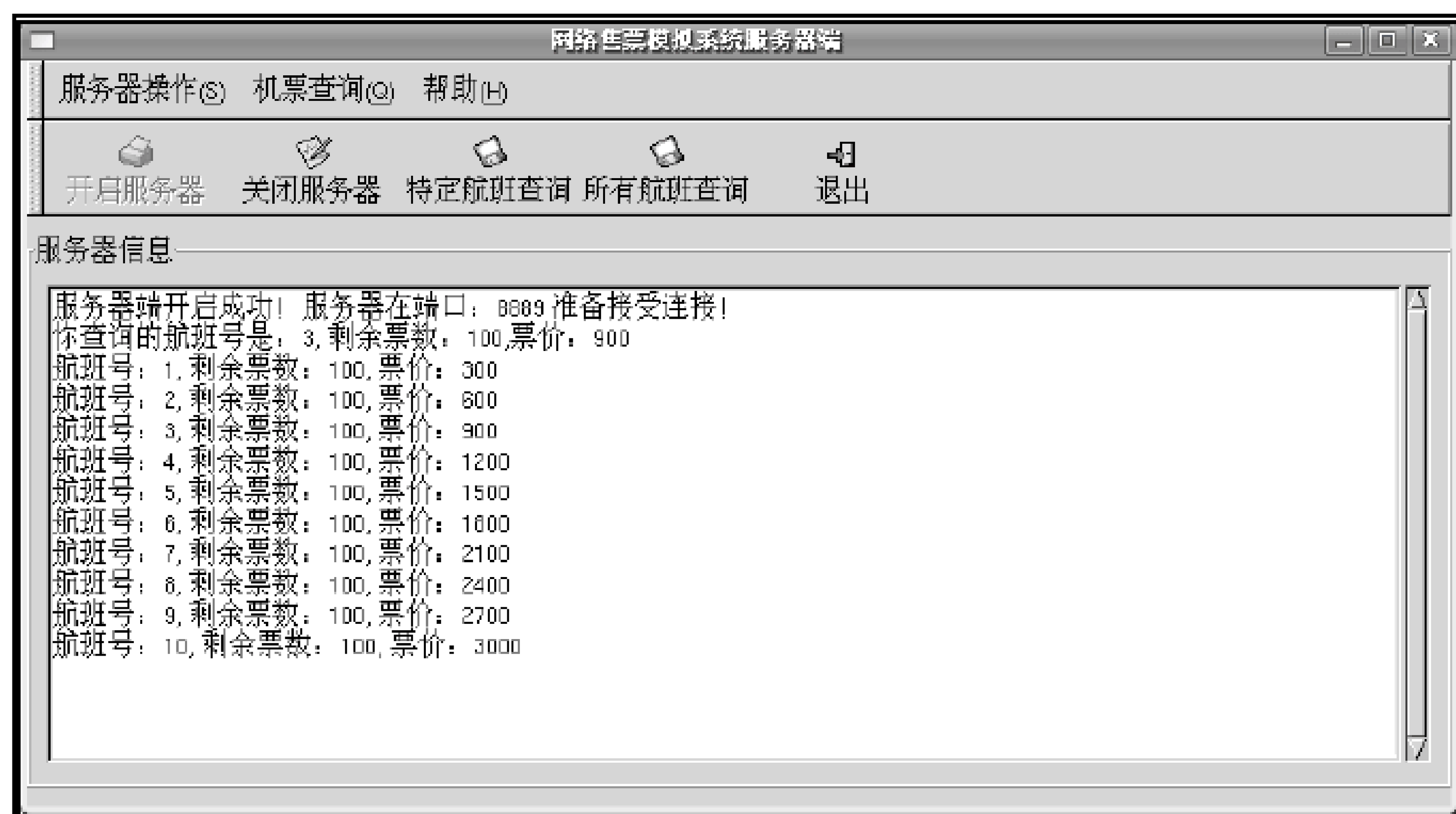


图 11-9　查询所有航班输出结果

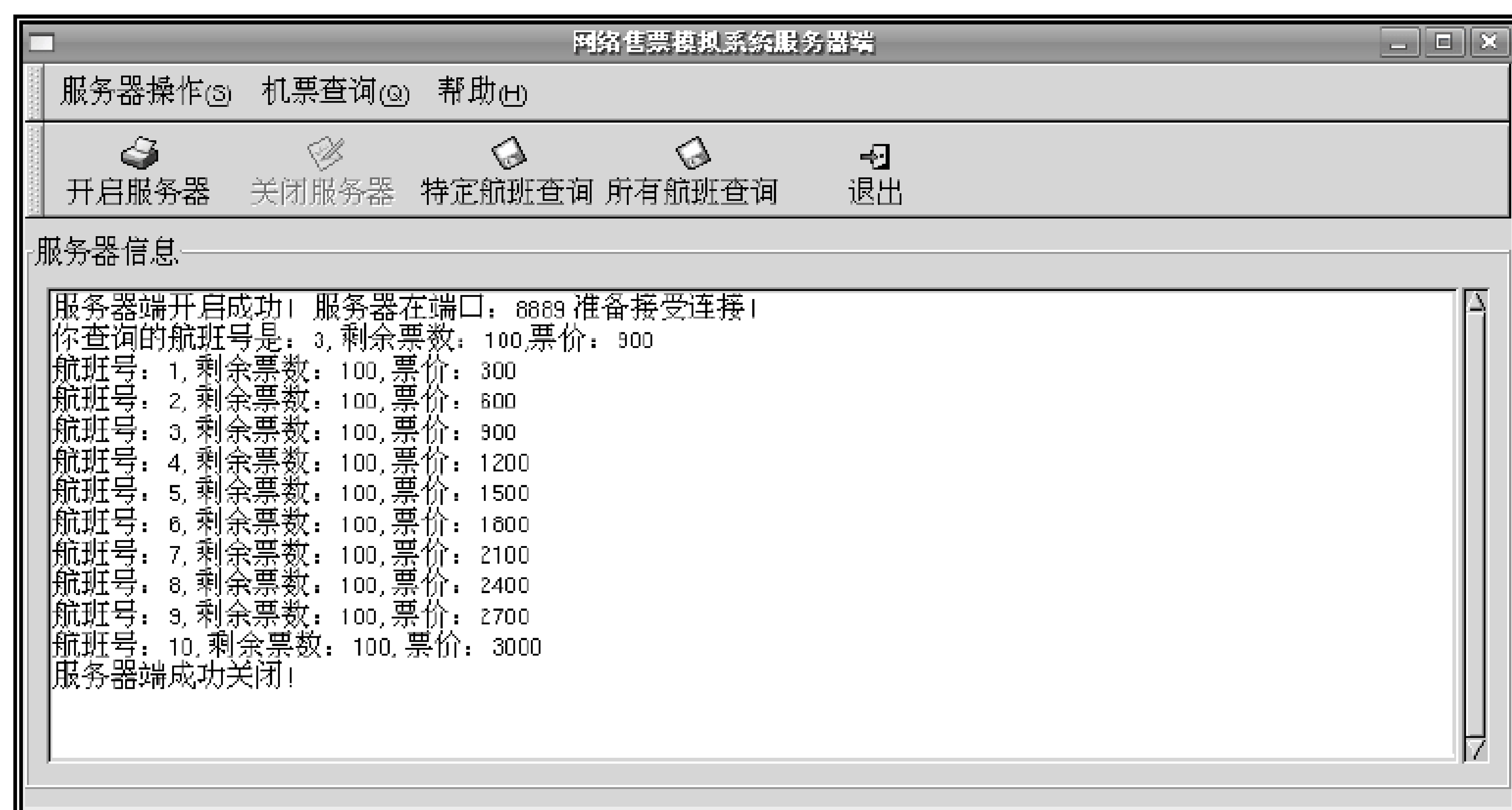


图 11-10　关闭服务器

单击“退出”按钮，即可退出程序。

在“帮助”菜单中，有操作的简单帮助和“关于”说明，读者单击即可查看，此处不再演示。

以上简单介绍了服务器端程序的功能。下面将介绍客户端程序代码和功能。

11.2.2　客户端源代码

客户端代码包括 globalapi.h、client.c 等文件构成。其中，globalapi.h 文件与服务器端相同，下面给出 client.c 文件：

client.c 的代码如下所示：

```
1    /*client.c*/
2
3    #include "globalapi.h"
4
5    int socket_fd;     //连接 socket
6    struct sockaddr_in server;                //服务器地址信息
7    int ret,i;
8    int flag=1;
9
10   static GtkWidget       *app;       /*程序主窗口*/
11   static GtkWidget *frame, *vbox, *box2 ,*table; /*box2 用来封装文本构件与垂直滚动条*/
12   static GtkWidget *clientwindow , *vscrollbar; //客户端窗口，用来输出相关提示信息
13
14   int isconnected=FALSE;            //是否已连接服务器
15
16   void connectserver(GtkWidget *widget, gpointer data);
17   void disconnect(GtkWidget *widget, gpointer data);
18   void buyticket(GtkWidget *widget,gpointer data);
19   void inquireone();
20   void inquireall();
21   void displaycontents(GtkWidget *widget, gpointer data);
22   void about(GtkWidget *widget, gpointer data);
23
24   /*生成客户端操作菜单项*/
25   GnomeUIInfo client_operation_menu[]={
26         GNOMEUIINFO_ITEM_NONE("连接服务器" ,"与远程服务器建立连接", connectserver),
27         GNOMEUIINFO_ITEM_NONE("断开连接" ,"断开与远程服务器的连接", disconnect),
28         GNOMEUIINFO_ITEM_NONE("购买机票" ,"购买机票", buyticket),
29         GNOMEUIINFO_ITEM_NONE("退出" ,"退出程序", gtk_main_quit),
30         GNOMEUIINFO_END
31   };
32   /*生成航班查询菜单项*/
33   GnomeUIInfo inquire_menu[]={
34         GNOMEUIINFO_ITEM_NONE("查询特定航班" ,"查询某一特定航班机票信息", inquireone),
35         GNOMEUIINFO_ITEM_NONE("查询所有航班" ,"查询所有航班机票信息", inquireall),
36         GNOMEUIINFO_END
37   };
38   /*生成帮助菜单项*/
39   GnomeUIInfo help_menu[]={
40         GNOMEUIINFO_ITEM_NONE("显示内容" ,"显示帮助内容", displaycontents),
41         GNOMEUIINFO_ITEM_NONE("关于" ,"关于此程序说明", about),
```

```
42          GNOMEUIINFO_END
43      };
44      /*生成顶层菜单项*/
45      GnomeUIInfo menubar[]= {
46          GNOMEUIINFO_SUBTREE("客户端操作(_S)", client_operation_menu),
47          GNOMEUIINFO_SUBTREE("机票查询(_Q)", inquire_menu),
48          GNOMEUIINFO_SUBTREE("帮助(_H)", help_menu),
49          GNOMEUIINFO_END
50      };
51      /*生成工具栏*/
52      GnomeUIInfo toolbar[]= {
53          GNOMEUIINFO_ITEM_STOCK("连接服务器","与远程服务器建立连接",
                    connectserver, GNOME_STOCK_PIXMAP_PRINT),
54          GNOMEUIINFO_ITEM_STOCK("断开连接","断开与远程服务器的连接", disconnect,
                GNOME_STOCK_PIXMAP_CUT),
55          GNOMEUIINFO_ITEM_STOCK("购买机票","购买机票", buyticket,
                GNOME_STOCK_PIXMAP_OPEN),
56          GNOMEUIINFO_ITEM_STOCK("查询特定航班","查询某一特定航班机票信息",
                inquireone, GNOME_STOCK_PIXMAP_SAVE),
57          GNOMEUIINFO_ITEM_STOCK("查询所有航班","查询所有航班机票信息",
                inquireall, GNOME_STOCK_PIXMAP_SAVE),
58          GNOMEUIINFO_ITEM_STOCK("退出", "退出程序", gtk_main_quit,
                GNOME_STOCK_PIXMAP_EXIT),
59          GNOMEUIINFO_END
60      };
61
62      /*消息内容输出函数*/
63      void display_info(char *msg, GtkWidget *window)
64      {
65          gtk_text_freeze (GTK_TEXT (window));
66          gtk_text_insert (GTK_TEXT (window), NULL, &window->style->black, NULL,msg, -1);
67          gtk_text_thaw (GTK_TEXT (window));
68      }
69
70      /*连接服务器操作*/
71      void connectserver(GtkWidget *widget, gpointer data)
72      {
73          char msg[512];          //提示信息
74          GtkWidget *isenable;
75          int i;
76
77          if(!isconnected)
78          {
```

```
79		/*创建套接字*/
80		socket_fd=socket(AF_INET,SOCK_STREAM,0);
81		if(socket_fd<0) {
82			sprintf(msg,"创建套接字出错！  \n");
83			display_info(msg,clientwindow);
84			return;
85		}
86		/*设置接收、发送超时值*/
87		struct timeval time_out;
88		time_out.tv_sec=5;
89		time_out.tv_usec=0;
90		setsockopt(socket_fd, SOL_SOCKET, SO_RCVTIMEO, &time_out, sizeof(time_out));
91
92		/*填写服务器的地址信息*/
93		server.sin_family=AF_INET;
94		server.sin_addr.s_addr=inet_addr("127.0.0.1");//htonl(INADDR_ANY);
95		server.sin_port=htons(SERVER_PORT_NO);
96
97		/*连接服务器*/
98		ret=connect(socket_fd,(struct sockaddr*)&server, sizeof(server));
99		if(ret<0) {
100			sprintf(msg,"连接服务器 出错！ \n",SERVER_PORT_NO);
101			display_info(msg,clientwindow);
102			close(socket_fd);
103			return;
104		}
105
106		//成功后输出提示信息
107		sprintf(msg,"连接服务器成功！ \n");
108		display_info(msg,clientwindow);
109		isconnected=TRUE;
110
111		/*连接服务器菜单项和工具条灰化*/
112		isenable=toolbar[0].widget;
113		gtk_widget_set_sensitive(isenable,FALSE);
114		isenable=client_operation_menu[0].widget;
115		gtk_widget_set_sensitive(isenable,FALSE);
116
117		/*其他菜单项和工具条使能*/
118		for(i=1;i<=4;i++) {
119			isenable=toolbar[i].widget;
120			gtk_widget_set_sensitive(isenable,TRUE);
121		}
```

```
122                 for(i=1;i<=2;i++) {
123                         isenable=client_operation_menu[i].widget;
124                         gtk_widget_set_sensitive(isenable,TRUE);
125                 }
126                 isenable=menubar[1].widget;
127                 gtk_widget_set_sensitive(isenable,TRUE);
128
129         }
130 }
131
132 /*断开连接操作*/
133 void disconnect(GtkWidget *widget, gpointer data)
134 {
135         GtkWidget *isenable;
136         char msg[512];
137         if(isconnected)
138         {
139                 close(socket_fd);
140                 sprintf(msg,"断开连接成功！  \n");
141                 display_info(msg,clientwindow);
142                 isconnected=FALSE;
143
144                 /*连接服务器菜单项和工具条使能*/
145                 isenable=toolbar[0].widget;
146                 gtk_widget_set_sensitive(isenable,TRUE);
147                 isenable=client_operation_menu[0].widget;
148                 gtk_widget_set_sensitive(isenable,TRUE);
149
150                 /*其他菜单项和工具条灰化*/
151                 for(i=1;i<=4;i++) {
152                         isenable=toolbar[i].widget;
153                         gtk_widget_set_sensitive(isenable,FALSE);
154                 }
155                 for(i=1;i<=2;i++) {
156                         isenable=client_operation_menu[i].widget;
157                         gtk_widget_set_sensitive(isenable,FALSE);
158                 }
159                 isenable=menubar[1].widget;
160                 gtk_widget_set_sensitive(isenable,FALSE);
161         }
162 }
163 /*购买机票对话框的文本输入框，用来获取输入的航班号和票数*/
164 struct flight_entry_t {
```

```
165         GtkWidget *flight_ID;
166         GtkWidget *ticket_num;
167     } st_flight;
168
169     /*购买机票的按钮回调函数*/
170     int button_buyticket(GnomeDialog *dialog, gint id,gpointer data)
171     {
172         char msg[512];
173         char send_buf[512],recv_buf[512];
174         GtkWidget *mbox;
175         if(id==0) {                          //如果"确定"按钮被单击
176             struct flight_entry_t *p=(struct flight_entry_t *)data;
177             GtkWidget *flight_ID_entry=p->flight_ID;
178             GtkWidget *ticket_num_entry=p->ticket_num;
179             /*获取输入的航班号*/
180             char *str=gtk_entry_get_text(GTK_ENTRY(flight_ID_entry));
181             int flight_ID=atoi(str);
182             if(flight_ID<=0 || flight_ID>10) { //判断输入的航班号是否正确，不正确的话，
                  给出提示信息，重新输入
183                 mbox = gnome_message_box_new ("输入的航班号错误！请重新输入！",
                        GNOME_MESSAGE_BOX_INFO,
                        GNOME_STOCK_BUTTON_OK,NULL ) ;
184                 gtk_widget_show (mbox);
185                 gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
186                 gnome_dialog_set_parent(GNOME_DIALOG
                     ( mbox),GTK_WINDOW(dialog));
187                 gtk_entry_set_text(GTK_ENTRY(flight_ID_entry)," ");
188                 return ;
189             }
190             /*获取输入的机票数*/
191             str=gtk_entry_get_text(GTK_ENTRY(ticket_num_entry));
192             int ticket_num=atoi(str);
193             if(ticket_num<=0) {    //判断输入的票数是否正确,不正确的话,给出提示信息,
                                       重新输入
194                 mbox = gnome_message_box_new ("输入的票数错误！请重新输入！",
                        GNOME_MESSAGE_BOX_INFO,
                        GNOME_STOCK_BUTTON_OK,NULL ) ;
195                 gtk_widget_show (mbox);
196                 gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
197                 gnome_dialog_set_parent(GNOME_DIALOG
                     ( mbox),GTK_WINDOW(dialog));
198                 gtk_entry_set_text(GTK_ENTRY(ticket_num_entry)," ");
199                 return ;
```

```
200            }
201            /*购买机票*/
202
203            init_message();
204            message.msg_type=BUY_TICKET;
205            message.flight_ID=flight_ID;
206            message.ticket_num=ticket_num;
207            memcpy(send_buf,&message,sizeof(message));
208            int ret=send(socket_fd, send_buf,sizeof(message),0);
209            /*发送出错*/
210            if(ret==-1) {
211                mbox = gnome_message_box_new ("发送失败！请重新发送！",
                        GNOME_MESSAGE_BOX_INFO,
                        GNOME_STOCK_BUTTON_OK,NULL ) ;
212                gtk_widget_show (mbox);
213                gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
214                gnome_dialog_set_parent(GNOME_DIALOG
                 ( mbox),GTK_WINDOW(dialog));
215                return ;
216            }
217            ret=recv(socket_fd,recv_buf,sizeof(message),0);
218            if(ret==-1) {
219                mbox = gnome_message_box_new ("接收失败！请重新发送！",
                        GNOME_MESSAGE_BOX_INFO,
                        GNOME_STOCK_BUTTON_OK,NULL ) ;
220                gtk_widget_show (mbox);
221                gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
222                gnome_dialog_set_parent(GNOME_DIALOG
                 ( mbox),GTK_WINDOW(dialog));
223                return ;
224            }
225            memcpy(&message,recv_buf,sizeof(message));
226            if(message.msg_type==BUY_SUCCEED)
227                sprintf(msg,"购买成功！航班号：%d，票数：%d，总票价：
                     %d\n",message.flight_ID,message.ticket_num, message.ticket_total_price);
228            else
229                sprintf(msg,"购买失败！航班号：%d，剩余票数：%d，请求票数：
                     %d\n",message.flight_ID,message.ticket_num,ticket_num);
230            mbox = gnome_message_box_new (msg,GNOME_MESSAGE_BOX_INFO,
                   GNOME_STOCK_BUTTON_OK,NULL ) ;
231                gtk_widget_show (mbox);
232                gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
233                gnome_dialog_set_parent(GNOME_DIALOG
```

```
                    ( mbox),GTK_WINDOW(dialog));
234         }
235         else
236             gnome_dialog_close(dialog);
237
238     }
239
240     /*购买机票回调函数*/
241     void buyticket(GtkWidget *widget, gpointer data)
242     {
243         GtkWidget *dialog;
244         GtkWidget *label,*flight_entry;
245         dialog = gnome_dialog_new(_("购买机票"),GNOME_STOCK_BUTTON_OK ,
                  GNOME_STOCK_BUTTON_CANCEL ,NULL) ;
246
247         label=gtk_label_new("请输入要购买的航班号(1-10)：");
248         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),label,TRUE,TRUE ,0 ) ;
249         gtk_widget_show (label);
250
251         flight_entry=gtk_entry_new();
252         st_flight.flight_ID=flight_entry;
253         gtk_entry_set_visibility(GTK_ENTRY(flight_entry),TRUE);
254         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),flight_entry,
              FALSE, FALSE,0);
255         gtk_widget_show (flight_entry);
256
257         label=gtk_label_new("请输入要购买的机票数：");
258         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),label,TRUE,TRUE ,0 ) ;
259         gtk_widget_show (label);
260
261         flight_entry=gtk_entry_new();
262         st_flight.ticket_num=flight_entry;
263         gtk_entry_set_visibility(GTK_ENTRY(flight_entry),TRUE);
264         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),flight_entry,
                FALSE, FALSE,0);
265         gtk_widget_show (flight_entry);
266         gnome_dialog_set_default(GNOME_DIALOG(dialog),0);   //设定确定作为缺省按钮
267
268         gtk_signal_connect(GTK_OBJECT(dialog),"clicked" ,GTK_SIGNAL_FUNC(button_bu
                    yticket) ,&st_flight) ; //设定"确定""取消"按钮的回调函数
269         gtk_window_set_modal (GTK_WINDOW (dialog), TRUE);
270         gtk_widget_show (dialog);
271         gnome_dialog_set_parent(GNOME_DIALOG( dialog),GTK_WINDOW(app));
```

```
                //设定对话框的父窗口为主程序窗口
272
273  }
274  /*查询特定航班对话框的“确定”“取消”按钮的回调函数*/
275  int button_inquireone(GnomeDialog *dialog, gint id,gpointer data)
276  {
277      GtkWidget *flight=data;
278      GtkWidget *mbox;
279      char msg[512];
280      char send_buf[512],recv_buf[512];
281      int   flight_ID;
282      if(id==0) {                  //如果“确定”按钮被点击
283          sprintf(msg,gtk_entry_get_text(GTK_ENTRY(flight)));
284          flight_ID=atoi(msg);
285          if(flight_ID<=0 || flight_ID>10) { //判断输入的航班号是否正确，不正确的话，
             给出提示信息，重新输入
286              //gnome_dialog_close(dialog);
287              mbox = gnome_message_box_new ("输入的航班号错误！请重新输入！",
                     GNOME_MESSAGE_BOX_INFO,
                     GNOME_STOCK_BUTTON_OK,NULL ) ;
288              gtk_widget_show (mbox);
289              gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
290              gnome_dialog_set_parent(GNOME_DIALOG( mbox),GTK_WINDOW(dialog));
291              gtk_entry_set_text(GTK_ENTRY(flight)," ");
292              return ;
293          }
294          init_message();
295          message.msg_type=INQUIRE_ONE;
296          message.flight_ID=flight_ID;
297          memcpy(send_buf,&message,sizeof(message));
298          int ret=send(socket_fd, send_buf,sizeof(message),0);
299          /*发送出错*/
300          if(ret==-1) {
301              mbox = gnome_message_box_new ("发送失败！请重新发送！",
                     GNOME_MESSAGE_BOX_INFO,
                     GNOME_STOCK_BUTTON_OK,NULL ) ;
302              gtk_widget_show (mbox);
303              gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
304              gnome_dialog_set_parent(GNOME_DIALOG( mbox),GTK_WINDOW(dialog));
305              return ;
306          }
307          ret=recv(socket_fd,recv_buf,sizeof(message),0);
308          if(ret==-1) {
```

```
309                     mbox = gnome_message_box_new ("接收失败！请重新连接服务器！
                                \n",GNOME_MESSAGE_BOX_INFO,
                                GNOME_STOCK_BUTTON_OK,NULL ) ;
310                     gtk_widget_show (mbox);
311                     gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
312                     gnome_dialog_set_parent(GNOME_DIALOG( mbox),GTK_WINDOW(dialog));
313                     return ;
314               }
315               memcpy(&message,recv_buf,sizeof(message));
316               if(message.msg_type==INQUIRE_SUCCEED)
317                     sprintf(msg,"查询成功！航班号：%d，剩余票数：%d，票价：
                           %d\n",message.flight_ID,message.ticket_num, message.ticket_total_price);
318               else
319                     sprintf(msg,"查询失败！航班号：%d，剩余票数：未知\n",message.flight_ID);
320               display_info(msg,clientwindow);
321               gnome_dialog_close(dialog);
322         }
323         else                  //取消按钮被按下
324               gnome_dialog_close(dialog);
325   }
326
327   /*查询某一特定航班*/
328   void inquireone()
329   {
330         /**/
331         //int flight=toolbar[2].
332         GtkWidget *dialog;
333         GtkWidget *label,*flight_entry;
334         dialog = gnome_dialog_new(_("查询特定航班机票信息"),
                      GNOME_STOCK_BUTTON_OK ,
                      GNOME_STOCK_BUTTON_CANCEL ,NULL) ;
335
336         label=gtk_label_new("请输入要查询的航班号(1-10)：");
337         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),label,TRUE,TRUE ,0 ) ;
338         gtk_widget_show (label);
339
340         flight_entry=gtk_entry_new();
341         gtk_entry_set_visibility(GTK_ENTRY(flight_entry),TRUE);
342         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),flight_entry,
                       FALSE, FALSE,0);
343         gtk_widget_show (flight_entry);
344
345         gnome_dialog_set_default(GNOME_DIALOG(dialog),0);     //设定确定作为默认按钮
```

```
346
347         gtk_signal_connect(GTK_OBJECT(dialog),"clicked" ,GTK_SIGNAL_FUNC(button_inq
                                 uireone) ,flight_entry) ; //设定“确定” “取消”按钮的回调函数
348         gtk_window_set_modal (GTK_WINDOW (dialog), TRUE);
349         gtk_widget_show (dialog);
350         gnome_dialog_set_parent(GNOME_DIALOG( dialog),GTK_WINDOW(app));   //设定
                                 对话框的父窗口为主程序窗口
351 }
352
353 void inquireall()
354 {
355
356         int i,pos;
357         char msg[512];
358         char send_buf[512], recv_buf[512];
359         GtkWidget *mbox;
360         init_message();
361         message.msg_type=INQUIRE_ALL;
362         memcpy(send_buf,&message,sizeof(message));
363         int ret=send(socket_fd, send_buf,sizeof(message),0);
364         /*发送出错*/
365         if(ret==-1) {
366                 mbox = gnome_message_box_new ("发送失败！请重新发送！",GNOME_
                         MESSAGE_BOX_INFO,GNOME_STOCK_BUTTON_OK,NULL ) ;
367                 gtk_widget_show (mbox);
368                 gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
369                 gnome_dialog_set_parent(GNOME_DIALOG( mbox),GTK_WINDOW(app));
370                 return ;
371         }
372         ret=recv(socket_fd,recv_buf,sizeof(recv_buf),0);
373         if(ret==-1) {
374                 mbox = gnome_message_box_new ("接收失败！请重新发送！",GNOME_
                         MESSAGE_BOX_INFO,GNOME_STOCK_BUTTON_OK,NULL ) ;
375                 gtk_widget_show (mbox);
376                 gtk_window_set_modal (GTK_WINDOW (mbox), TRUE);
377                 gnome_dialog_set_parent(GNOME_DIALOG( mbox),GTK_WINDOW(app));
378                 return ;
379         }
380         pos=0;
381         sprintf(msg,"查询所有航班结果：\n");
382         display_info(msg,clientwindow);
383         for (i=0;i<ret;i=i+sizeof(message)) {
384                 memcpy(&message,recv_buf+pos,sizeof(message));
```

```
385                 if(message.msg_type==INQUIRE_SUCCEED)
386                     sprintf(msg,"查询成功！航班号：%d，剩余票数：%d，票价：
                        %d\n",message.flight_ID,message.ticket_num, message.ticket_total_price);
387                 else
388                     sprintf(msg,"查询失败！航班号：%d，剩余票数：未知\n",message.flight_ID);
389                 display_info(msg,clientwindow);
390                 pos+=sizeof(message);
391             }
392     }
393
394     /*帮助对话框和关于对话框的确定按钮处理函数*/
395     void dialog_ok(GnomeDialog *dialog, gint id,gpointer data)
396     {
397             gnome_dialog_close(dialog);
398     }
399
400     /*帮助———显示内容*/
401     void displaycontents(GtkWidget *widget, gpointer data)
402     {
403             GtkWidget *dialog;
404             GtkWidget *label;
405             char msg[5][512]={"连接服务器：与远程服务器建立连接","断开连接：断开与远程服
                            务器的连接","购买机票：购买机票","特定航班查询：查询某一特
                            定航班机票信息","所有航班查询：查询所有航班机票信息"};
406             dialog = gnome_dialog_new(_("帮助"),GNOME_STOCK_BUTTON_OK   ,NULL ,NULL) ;
407             for(i=0;i<5;i++) {
408                 label=gtk_label_new(msg[i]);
409                 gtk_box_pack_start(GTK_BOX(GNOME_DIALOG
                        (dialog)->vbox),label,TRUE,TRUE ,0 ) ;
410                 gtk_widget_show (label);
411             }
412
413             gtk_signal_connect(GTK_OBJECT(dialog),"clicked" ,GTK_SIGNAL_FUNC(dialog_ok)
                        ,&dialog) ;
414             gtk_window_set_modal (GTK_WINDOW (dialog), TRUE);
415             gtk_widget_show (dialog);
416             gnome_dialog_set_parent(GNOME_DIALOG( dialog),GTK_WINDOW(app));
417     }
418
419
420     /*帮助———关于*/
421     void about(GtkWidget *widget, gpointer data)
422     {
```

```
423         GtkWidget *dialog;
424         GtkWidget *contentlabel,*versionlabel,*authorlabel,*copyrightlabel;
425         dialog = gnome_dialog_new(_("关于本程序"),
                    GNOME_STOCK_BUTTON_OK,NULL ,NULL) ;
426         contentlabel=gtk_label_new(_("网络售票模拟系统客户端"));
427         versionlabel=gtk_label_new(_("版本：0.1"));
428         copyrightlabel=gtk_label_new(_("Copyright 2009"));
429         authorlabel=gtk_label_new(_("lxy"));
430         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),contentlabel,TRUE,
                              TRUE ,0 ) ;
431         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),versionlabel,TRUE,
                              TRUE ,0 ) ;
432         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),copyrightlabel,TRU
                              E,TRUE ,0 ) ;
433         gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dialog)->vbox),authorlabel,TRUE,
                              TRUE ,0 ) ;
434
435         gtk_widget_show (contentlabel);
436         gtk_widget_show (versionlabel);
437         gtk_widget_show (copyrightlabel);
438         gtk_widget_show (authorlabel);
439
440         gtk_signal_connect(GTK_OBJECT(dialog),"clicked" ,GTK_SIGNAL_FUNC(dialog_ok)
                              ,&dialog) ;
441         gtk_window_set_modal (GTK_WINDOW (dialog), TRUE);
442         gtk_widget_show (dialog);
443         gnome_dialog_set_parent(GNOME_DIALOG( dialog),GTK_WINDOW(app));
444     }
445
446     int main(int argc, char *argv[])
447     {
448         GtkWidget       *isenable;
449         int i;
450
451         //界面初始化部分
452         gtk_set_locale();
453         gnome_init("example", "0.1", argc, argv);
454         app=gnome_app_new("example", "网络售票模拟系统客户端");
455         gtk_signal_connect(GTK_OBJECT(app), "delete_event",
                   GTK_SIGNAL_FUNC(gtk_main_quit), NULL);
456         //添加菜单和工具栏
457         gnome_app_create_menus(GNOME_APP(app),menubar);
458         gnome_app_create_toolbar(GNOME_APP(app),toolbar);
```

```
459
460        /*启动程序时使工具条上断开连接、机票购买、机票查询以及相应的菜单项灰化*/

461        if(!isconnected) {
462                for(i=1;i<=4;i++) {
463                        isenable=toolbar[i].widget;
464                        gtk_widget_set_sensitive(isenable,FALSE);
465                }
466                for(i=1;i<=2;i++) {
467                        isenable=client_operation_menu[i].widget;
468                        gtk_widget_set_sensitive(isenable,FALSE);
469                }
470                isenable=menubar[1].widget;
471                gtk_widget_set_sensitive(isenable,FALSE);
472        }
473
474        gtk_window_set_default_size((GtkWindow *)app, 800, 600);
475        vbox=gtk_vbox_new(FALSE,0);
476        gnome_app_set_contents(GNOME_APP(app),vbox);
477
478        /*创建客户端输出信息窗口*/
479        frame=gtk_frame_new(NULL);
480        gtk_frame_set_label(GTK_FRAME(frame),"客户端信息");
481        gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT);
482
483        gtk_box_pack_start(GTK_BOX(vbox), frame, TRUE,TRUE,10);       /*最后一个参数控
                                                                            制间隔*/
484        gtk_widget_show(vbox);
485
486        box2 = gtk_vbox_new (FALSE, 10);
487        gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
488        gtk_container_add(GTK_CONTAINER(frame),box2);
489        gtk_widget_show (box2);
490
491        table = gtk_table_new (2, 2, FALSE);
492        gtk_table_set_row_spacing (GTK_TABLE (table), 0, 2);
493        gtk_table_set_col_spacing (GTK_TABLE (table), 0, 2);
494        gtk_box_pack_start (GTK_BOX (box2), table, TRUE, TRUE, 0);
495        gtk_widget_show (table);
496        //创建文本信息输出框
497        clientwindow=gtk_text_new (NULL, NULL);
498        gtk_text_set_editable (GTK_TEXT (clientwindow), FALSE);    //文本信息不可编辑
499        gtk_text_set_word_wrap(GTK_TEXT(clientwindow),TRUE );  //自动换行
```

```
500         gtk_table_attach (GTK_TABLE (table), clientwindow, 0, 1, 0, 1, GTK_EXPAND |
                GTK_SHRINK | GTK_FILL,
501         GTK_EXPAND | GTK_SHRINK | GTK_FILL, 0, 0);
502         gtk_widget_show (clientwindow);
503
504         vscrollbar = gtk_vscrollbar_new (GTK_TEXT (clientwindow)->vadj);
505         gtk_table_attach (GTK_TABLE (table), vscrollbar, 1, 2, 0, 1,GTK_FILL,
                GTK_EXPAND | GTK_SHRINK | GTK_FILL, 0, 0);
506         gtk_widget_show (vscrollbar);
507         gtk_widget_realize (clientwindow);
508
509         gtk_widget_show_all(app);
510         gtk_main();
511
512         return 0;
513     }
```

说明：client.c 是客户端主程序文件。程序第 5~14 行定义了相关的变量，包括连接socket、服务器地址信息、程序主窗口、客户端界面信息输出窗口等。第 25~60 行定义了界面的菜单和工具栏，如图 11-11 所示。第 63~444 行定义了菜单和工具栏相应的处理函数。关于函数的具体实现请参考程序注释。

第 446~514 行是服务器程序的 main 函数，在 main 函数中完成界面初始化(第 452~455 行)，生成菜单和工具栏(第 457~458 行)，并设置程序刚启动时菜单和工具栏的状态(第 461~472 行)，定义程序启动时窗口大小(第 474~476 行)，随后创建客户端信息输出窗口(第 479~507 行)，最后是显示所有窗口，进入 gtk 循环(第 509~510 行)。关于程序详细说明，请参考注释。

客户端编译命令如下：

```
$ gcc globalapi.h client.c -o client `gnome-config --cflags --libs gnomeui`
```

对服务器和客户端源代码分别编译连接成功后，可以在同一台机器或不同的主机上分别运行服务器程序和客户端程序。

下面我们从同一台机器上同时运行服务器进程和客户端进程来验证程序的功能。首先执行服务器端程序，然后执行客户端程序。

客户端程序执行结果如图 11-11 所示。

程序有 3 个顶层菜单和 5 个常用工具按钮。“客户操作”菜单包括“连接服务器”、“断开连接”、“购买机票”以及“退出”4 项。它们与工具栏中的对应项功能是相同的。“机票查询”菜单包括“特定航班查询”、“所有航班查询”，它们与工具栏的对应项功能相同。以下操作以工具栏为例进行说明，这些操作也可通过选择相应的菜单项来完成，不再另行说明。在客户端程序刚启动时，只有“连接服务器”是激活的，而其他项都是不可用的。



图 11-11　客户端程序界面

单击“连接服务器”时，如果服务器端开启，则连接成功后，客户端界面输出如图 11-12 所示，服务器端界面输出如图 11-13 所示。

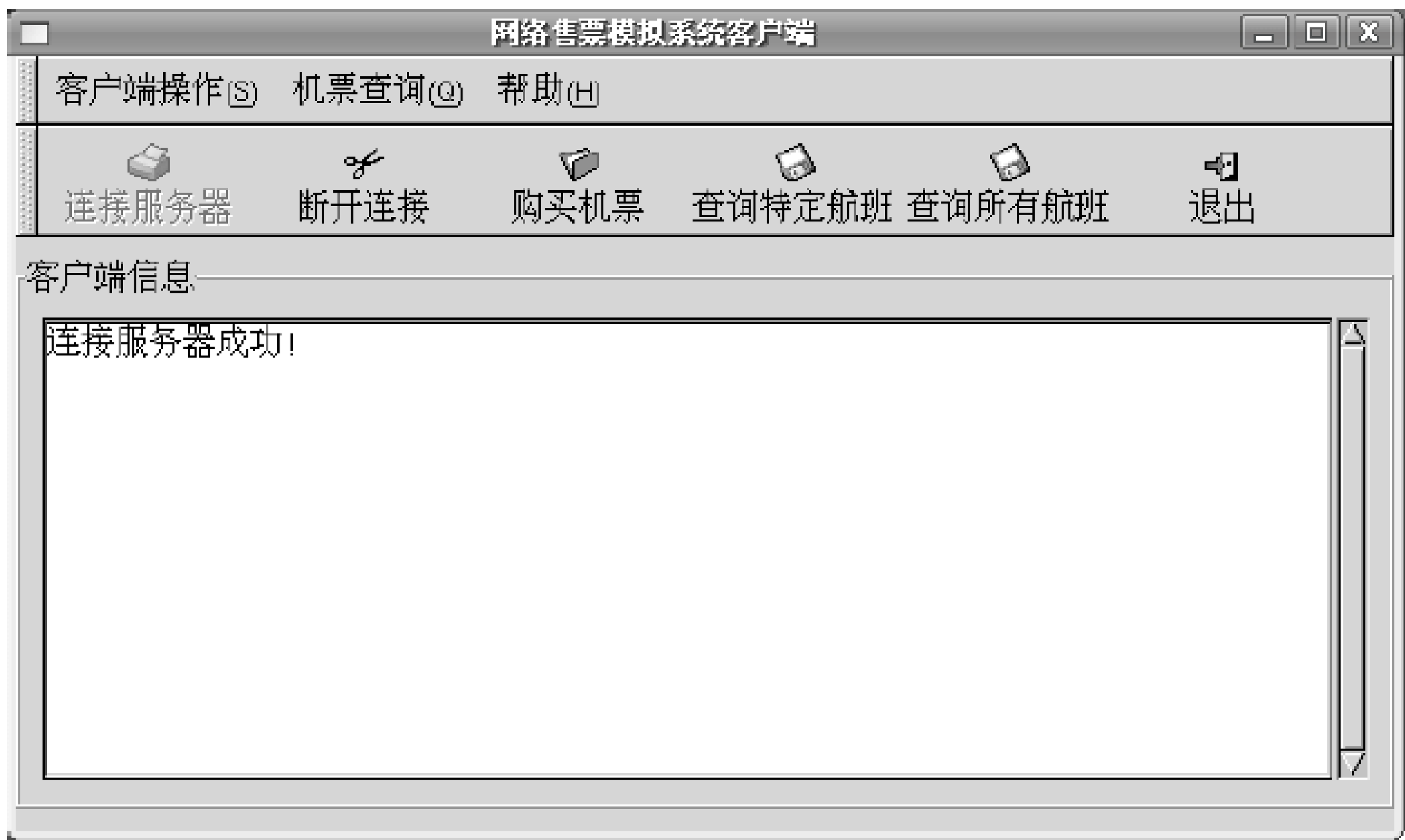


图 11-12　连接成功后，客户端输出

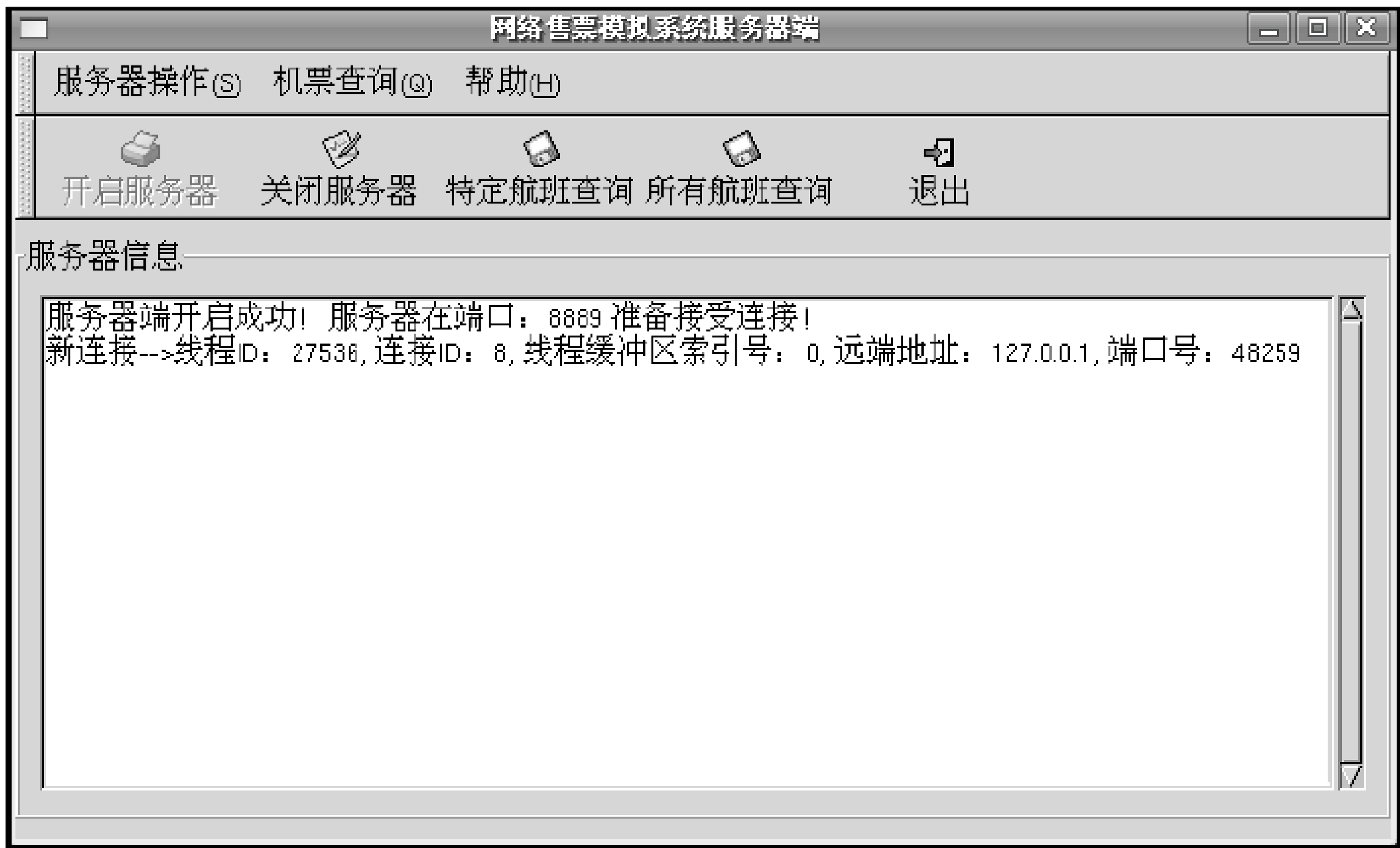


图 11-13　连接成功后，服务器端输出

连接成功后，客户端“连接服务器”项不可用，而其他项使能。此时，可以向服务器端发送请求。

点击“购买机票”，系统弹出如图 11-14 所示对话框。

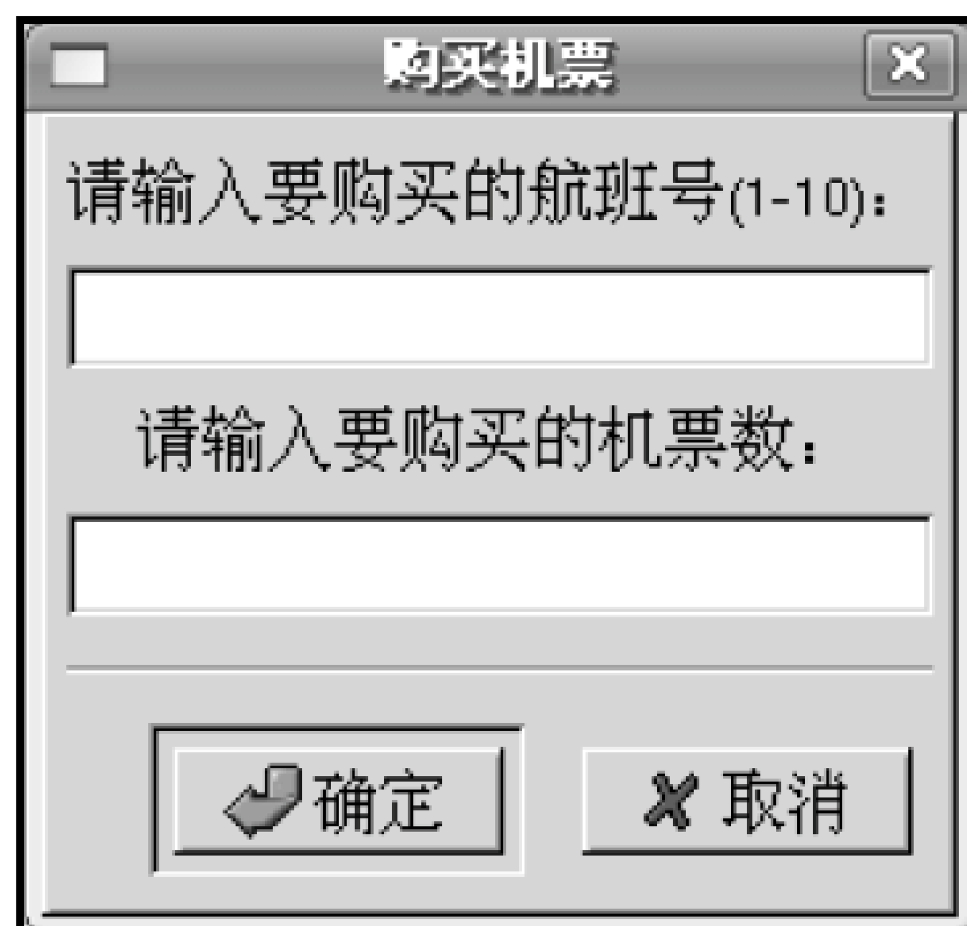


图 11-14　购买机票对话框

在输入框中输入航班号和机票数，如果航班号或机票数有错误，系统分别会给出提示，如图 11-15 所示。



图 11-15　错误提示

如果输入正确，并且成功后，客户端给出提示信息如图 11-16 所示。



图 11-16　购票成功客户端提示信息

此时，切换到服务器端程序，可以看到服务器端程序输出信息如图 11-17 所示。

从图 11-16 与图 11-17 可以看出，二者数据吻合，说明程序功能正确。

此时，再查看机票相关信息，在客户端程序单击“确定”按钮后，再点击“取消”按钮退出购买机票对话框，单击“查询特定航班”，在弹出的对话框中输入 6，单击“确定”按钮，可以看到查询结果如图 11-18 所示。

从图 11-18 可以看出，剩余机票数还有 66 张，这与刚才购买机票的信息符合。

此时，切换到服务器端程序，可以看到服务器端程序输出信息如图 11-19 所示。

如果要购买的票数大于剩余票数，则购买失败，客户端会给出相应的提示信息，读者可以进行验证，此处不再进行演示。

此外，“帮助”菜单内容与服务器端类似，读者运行一下即可看到结果，此处也不再进行演示。

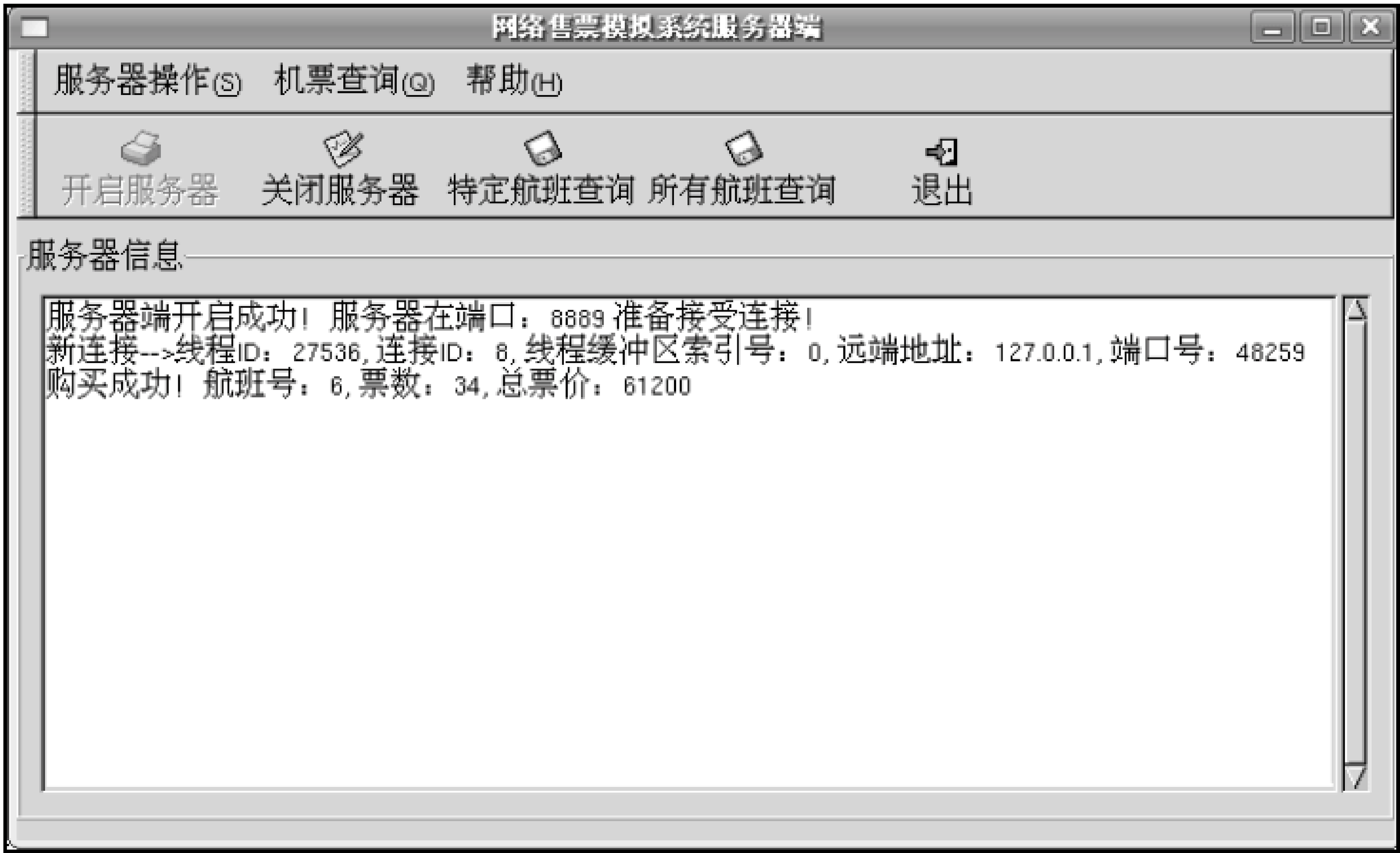


图 11-17　购票成功，服务器端输出信息

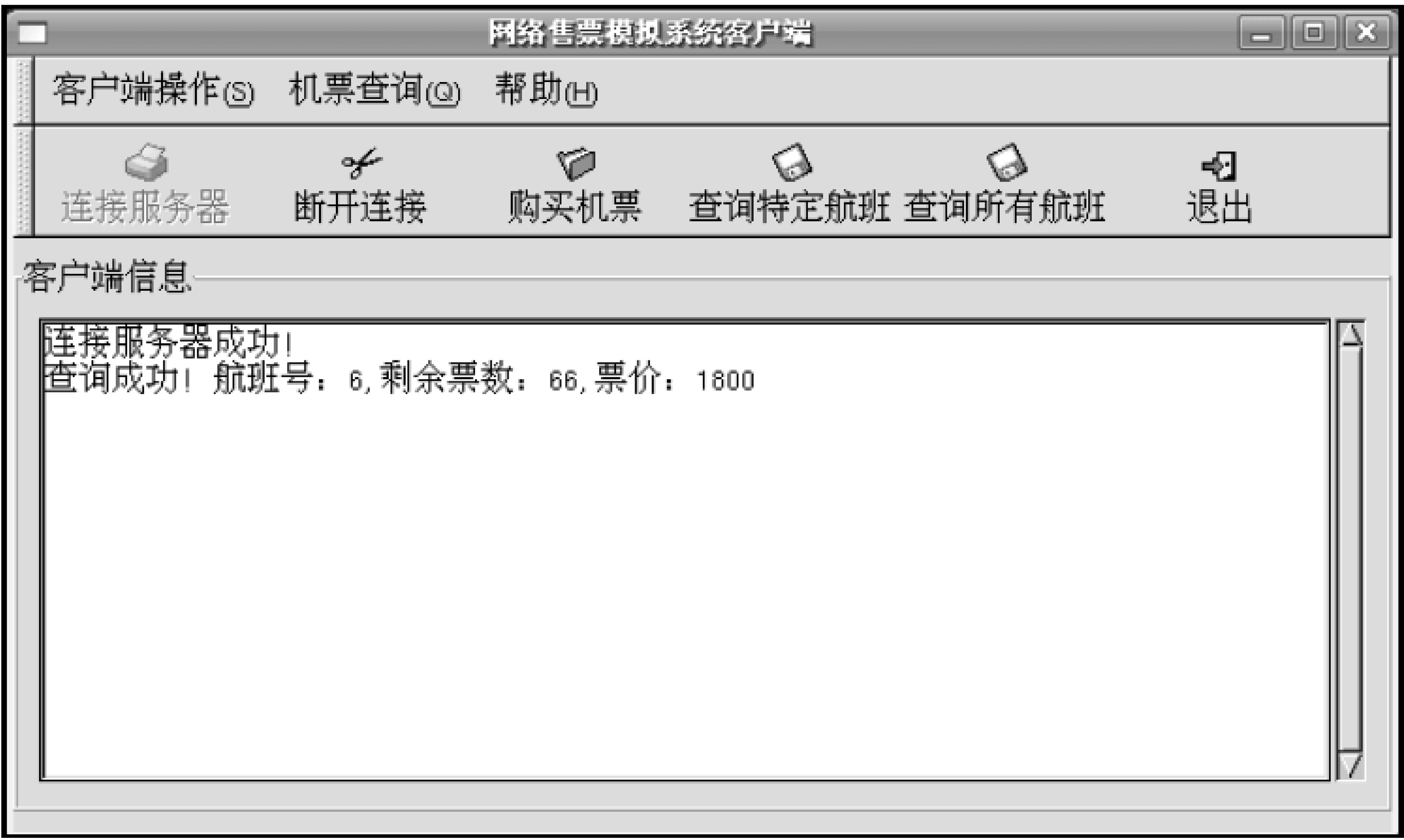


图 11-18　查询特定航班机票

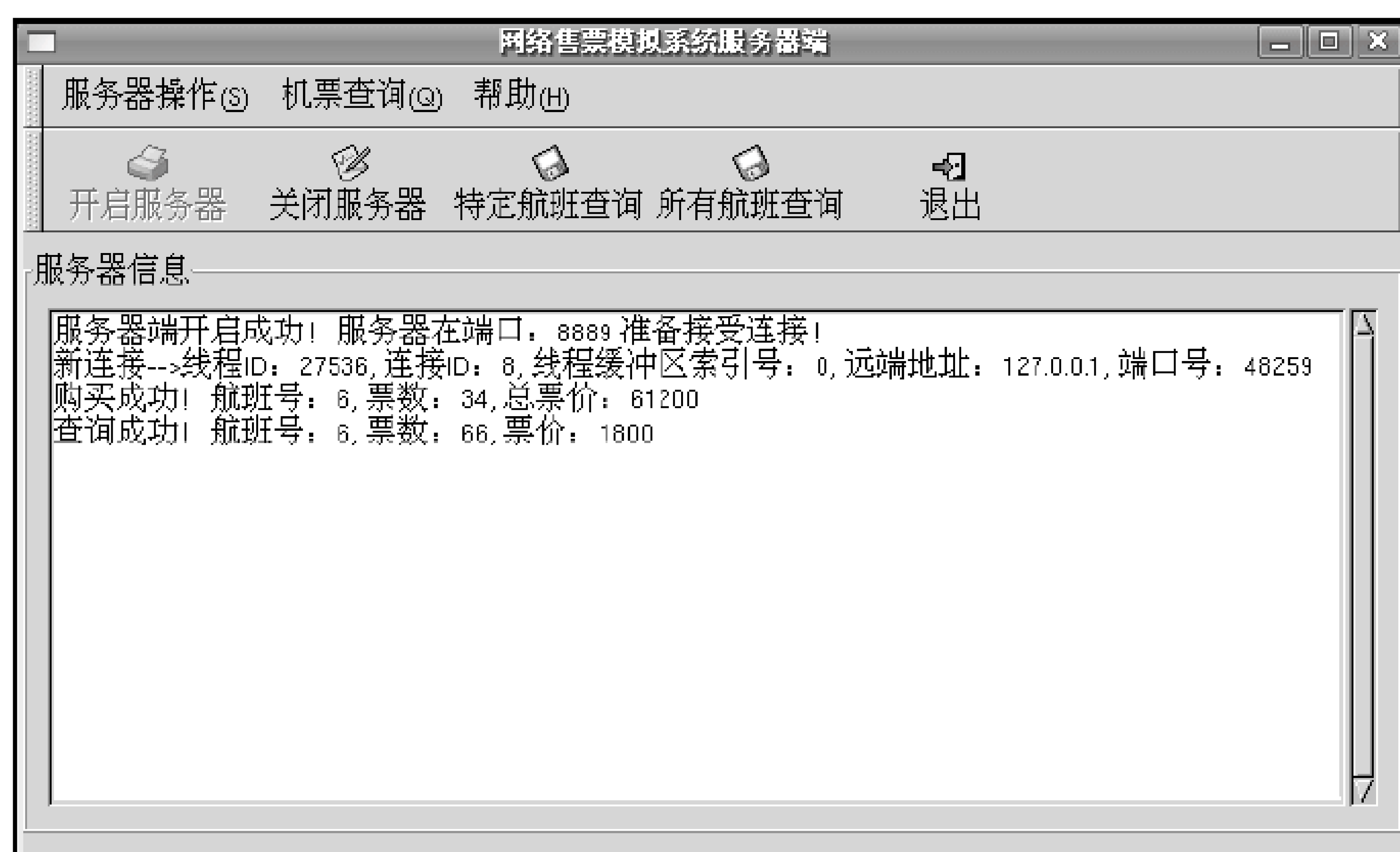


图 11-19　查询特定航班服务器端输出信息

以上就是一个网络售票系统的简单模拟，整个系统的框架已经建立，可以在此框架的基础上进一步完善，添加相应的功能，形成一个功能丰富，完整实用的网络售票系统。

11.3　小　结

本章通过示例，设计并实现了一个飞机票网络售票系统的模拟程序。首先说明系统的总体设计，主要包含通信消息格式的设计、服务器端的设计和客户端的设计。并给出了服务器端和客户端的程序源代码。读者在本章给出的示例基础上，可以进一步学习，逐渐掌握 Linux 下编写大型 C 程序的基本方法。

读者意见反馈卡

亲爱的读者：

感谢您购买了本书，希望它能为您的工作和学习带来帮助。为了今后能为您提供更优秀的图书，请您抽出宝贵的时间填写这份调查表，然后剪下寄到：北京清华大学出版社第五事业部(邮编 100084)；您也可以把意见反馈到 cwkbook@tup.tsinghua.edu.cn。邮购咨询电话：010-62786544，客服电话：010-62776969。我们将充分考虑您的意见和建议，并尽可能地给您满意的答复。谢谢！

本 书 名： ________________

个人资料： ________________

姓　　名： ________ **性　别：** □男 □女 **出生年月(或年龄)：** ________

文化程度： ________ **职　业：** ________ **通讯地址：** ________

电话(或手机)： ________ **传　真：** ________ **电子信箱(E-mail)：** ________

您是如何得知本书的： ________________

□别人推荐 □出版社图书目录 □网上信息 □书店

□杂志、报纸等的介绍(请指明)________ □其他(请指明)________

您从何处购得本书： □书店 □电脑商店 □软件销售处 □邮购 □商场 □其他

影响您购买本书的因素(可复选)：

□封面封底 □装帧设计 □价格 □内容提要、前言或目录 □书评广告

□出版社名声 □作者名声 □责任编辑

□其他：________________

您对本书封面设计的满意度： □很满意 □比较满意 □一般 □较不满意 □不满意 □改进建议________

您对本书印刷质量的满意度： □很满意 □比较满意 □一般 □较不满意 □不满意 □改进建议________

您对本书的总体满意度：

从文字角度：□很满意 □比较满意 □一般 □较不满意 □不满意

从技术角度：□很满意 □比较满意 □一般 □较不满意 □不满意

本书最令您满意的是：

□讲解浅显易懂 □内容充实详尽 □示例丰富到位 □指导明确合理 □其他：________

您希望本书在哪些方面进行改进？ ________________

您希望增加什么系列或软件的图书： ________________

您最希望学习的其他软件： 1.________ 2.________ 3.________ 4.________

您对使用中文版软件或外文版软件介意吗？更喜欢使用哪一种版本？

□介意 □无所谓 □中文版 □外文版

您对图书所用软件版本是否很介意？是否要求用最新版本？

□是，要求是最新版本 □无所谓 □不，因为硬件或软件跟不上要求

您是如何学习最新软件的？

□看计算机书 □看多媒体教学光盘 □自己摸索或查看软件的帮助信息 □参加培训班 □向其他人请教

□其他：________________

您的其他要求： ________________